André Greubel, Sven Strickroth, Michael Striewe (Hrsg.)

Proceedings of the Sixth Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2023)

12. und 13. Oktober 2023
München

In Zusammenarbeit mit der Fachgruppe Bildungstechnologien der GI e.V.

Vorwort

Auch nach Aufkommen von Unterstützungstools wie Co-Pilot ist der Erwerb eigenständiger Programmierfähigkeiten aus der informatischen Bildung nicht wegzudenken. Die bereits seit 2010 kontinuierlich stark steigende Anzahl an Studierenden stellt dabei weiterhin besondere Anforderungen an die Bewertung von in der Lehre eingesetzten Programmieraufgaben.

Auf dieser Basis findet in diesem Jahr bereits zum sechsten Mal der Workshop zur „Automatischen Bewertung von Programmieraufgaben“ statt; dieses Mal in München. Wie auch bei den vorherigen Auflagen stehen dabei Systeme zur Bewertung von Programmieraufgaben, Methoden zur automatischen Erstellung von Feedback, sowie Erfahrungswerte zur Plagiatserkennung im Zentrum.

Insgesamt wurden in diesem Jahr vierzehn Beiträge zur Begutachtung eingereicht. Aus diesen wurden schließlich elf Beiträge für die Präsentation auf dem Workshop ausgewählt. Erstmals wurde dabei ein Shepherding-Verfahren erprobt, in dem ausgewählte Beiträge vor der Einreichung der finalen Fassung intensiv durch die Vorsitzenden des Programmkomitees betreut wurden.

Durch die erfreulich hohe Zahl an angenommenen Beiträgen, konnte ein breit gefächertes Workshop-Programm erstellt werden: In einer Session werden Bewertungssysteme für Sprachen vorgestellt, die in den vergangenen Jahren noch wenig Aufmerksamkeit bekommen hatten (C, Haskell und Python). In einer weiteren Session werden Methoden diskutiert, mit denen durch Parametrisierung automatisch verschiedene Varianten ähnlicher Aufgaben erzeugt werden. Eine weitere Session thematisiert verschiedene Einsatzszenarien zur automatischen Bewertung. Weitere Beiträge im Workshop befassen sich mit Erfahrungswerten zur Plagiatserkennung und werfen einen Blick auf die Zusammensetzung der Forschungs-Community rund um die automatische Bewertung von Programmieraufgaben.

Ergänzt wird dieses Programm durch eine Keynote von Prof. Dr. Florian Zuleger, der als Fachwissenschaftler Einblicke in seine Forschung zur automatischen Analyse und Reparatur von Programmen gibt.

Wir bedanken uns bei allen Einreichenden und den Mitgliedern des Programmkomitees für die Erstellung bzw. das Review der Beiträge sowie allen Teilnehmenden des Workshops. Ohne dieses gemeinsame Engagement wäre der Workshop so nicht möglich!

André Greubel, Sven Strickroth, Michael Striewe
Oktober 2023

Organisation

André Greubel (Julius-Maximilians-Universität Würzburg)
Sven Strickroth (Ludwig-Maximilians-Universität München)
Michael Striewe (Universität Duisburg-Essen)

Programmkommitee

Torsten Brinda (Universität Duisburg-Essen)
Ralf Dörner (RheinMain University of Applied Sciences)
Robert Garmann (Hochschule Hannover)
Lukas Iffländer (Universität Würzburg)
Natalie Kiesler (dipf)
Tilman Michaeli (TU München)
Rainer Oechsle (Hochschule Trier)
Uta Priss (Ostfalia Hochschule für angewandte Wissenschaften)
Ulrik Schroeder (RWTH Aachen)

Weitere Reviews:

Annabell Brocker (RWTH Aachen)

Inhaltsverzeichnis

Algorithmenerkennung und Programmreparatur mittels Dynamischer Programmanalyse

Florian Zuleger[1]

Abstract: In diesem Vortrag beschreibe ich zwei Ansätze zur Erkennung der algorithmischen Idee eines Programms und ihre Anwendung in der Feedback-Generierung für einführende Programmieraufgaben. Beide Techniken basieren auf der dynamischen Programmanalyse, in Verbindung mit Constraintprogrammierung. Die erste Technik ist halbautomatisch und zielt darauf ab, Performanceprobleme in Programmen zu finden. Die zweite Technik verwendet einen großen Korpus korrekter Programme, die anhand von syntaktischen Kriterien und Programmausführungen geclustert und anschließend zur Reparatur fehlerhafter Programme verwendet werden.

[1] Institute of Logic and Computation, Technische Universität Wien, Austria, Florian.Zuleger@tuwien.ac.at

Automated Detection of Bugs in Error Handling for Teaching Secure C Programming

Max Schrötter[1], Maximilian Falk[1], Bettina Schnor[1]

Abstract: The Low-Level programming language C is widely used for Operating Systems, Embedded Systems and other performance critical applications. Since these applications are often security critical, they require secure programming. The C language on the other hand allows novice programmers to write insecure code easily. This makes it especially important to teach secure programming and give students feedback on potential security issues. One critical bug that is often overlooked is the incorrect handling of errors. In this paper, we present an Error Handling Analyzer (EHA) for the CoFee framework. The EHA detects missing error handling and incorrect error handling using the Clang Static Analyzer. We evaluated EHA on 100 student submissions and found that error handling bugs are a common mistake and that EHA can detect more than 80 % of the error handling bugs in these submissions.

Keywords: Automated Assessment; Continuous Integration; Continuous Feedback; Situated Learning; Secure Programming

1 Introduction

Automated feedback frameworks for programming courses have been extensively studied. A survey by Strickroth et al. from 2022 created a searchable Corpus[2] of 178 grading and feedback systems published till 2020 [SS22b]. Out of these 178 systems, 53 are reported to support the language C. Out of those only a few [Ch17; HK18; KS18] use modern software engineering workflows like version control and continuous integration for feedback, which promotes situated learning. Furthermore, most frameworks focus on functional correctness using unit tests and not on secure and robust programming. Some frameworks include diagnostics from analyzers or compilers, but are limited to only one tool. Artemis [KS18] for example includes diagnostics from GCC, while Canary [HK18] includes reports from Valgrind.

In our previous work [SS22a], we compiled a test suite of common mistakes described by the Secure Coding Standard of the SEI-CERT[3]. These were supplemented by our own teaching experience at the University of Potsdam. Using this test suite, we compared industry standard tools and the three overall winners of the International Competition on Software Verification SV-Comp 2022 [Be22]. The results showed that not a single tool can detect all mistakes. We therefore introduced the CoFee (Continuous Feedback) framework, which

[1] University of Potsdam, Department of Computer Science , An der Bahn 2, 14476 Potsdam, Germany firstname.lastname@uni-potsdam.de

[2] https://systemscorpus.strickroth.net/

[3] https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard

doi:10.18420/abp2023-1

uses a modular approach and integrates state of the art analyzers such as the Clang Static Analyzer[4] to give students fast, continuous and meaningful feedback, which is focused on secure and robust aspects of the analyzed source code [SS22a].

In C, functions usually return error codes via the return value and users should check this value to prevent undefined behavior to avoid security and reliability issues. Several Common Vulnerability and Exposures (CVEs) are showing that erroneous error handling may lead to security problems such as memory corruption, privilege escalation and Denial-of-Service attacks [Wu21]. The C standard however neither enforces this error checking, like in Rust, nor do the compilers provide diagnostics about missing error handling. Hence, the checking of correct error handling is up to the programmer to guarantee good software quality. Besides a simple checker for allocation functions only, no tools included in CoFee provided any feedback on missing error handling yet.

Therefore, the contributions of this paper are:

- We show that missing error handling is a common mistake in real world code and also in student submissions (see section 3).
- We present the new Error Handling Analyzer (EHA) for the CoFee framework (see section 4).
- We evaluate the EHA on student submissions and show that it can detect more than 80 % of the error handling bugs (see section 5).

2 Related Work

Several approaches to detect and fix missing error handling code have been proposed by researchers. Some have also been included in tools like the Clang static analyzer (CSA). For example the UncheckedReturn checker that is part of the CSA only checks for missing error handling for six functions that are used to drop privileges in the Glibc. For those functions the analysis is limited to the abstract syntax tree (AST). If the parent node of the function call is a compound statement, meaning the result of the function is neither saved to a variable nor compared to a value, a bug report is created. This however means that the checker does not verify whether actual errors are handled or whether the results are only stored in a variable and never checked.

The approach in EPEx [Ja16] and ErrDoc [TR17] use symbolic execution to detect error handling bugs using a predefined or inferred error specification by APEx [KRJ16]. Both use the CSA to identify possible error paths. A potential error path is a control flow path starting at the function call that might return errors and is defined in the error specification. The specification includes the function name and the return values that should be interpreted as errors. If the symbolic execution evaluates that the returned value includes the error values

[4] https://clang-analyzer.llvm.org/

specified by the specification, that path is marked as error path. EPEx then checks at exit calls, return statements and predefined logging functions whether on an error path values are e. g. returned that indicate no errors. ErrDoc adopts the approach of EPEx but extends some heuristics to detect error handling code. It also evaluates the AST and if the direct parent of the error generating function is an if statement, the code block in the if or else statement depending on symbolic execution is considered as error handling code. Error handling codes are also disregarded if they are longer than 5 statements.

These approaches however do not adopt well to novice student code. Generating diagnostics of error handling mistakes at return statements is too late. Since error handling needs to be done before the result of the function call is used and not when a return statement is reached. For example opening a file, reading from it and then checking for errors in the opening call is not detected by these approaches. Novice students also do not always handle errors directly under the function call. Although one may consider error handling code directly under the error generating statement is best practice, this is not always the case. The return value of `fork` is often first checked to differentiate between parent and child process and then checked for errors. Therefore, to keep false positives low, the checker should make no assumptions about the location of the error handling code.

The approach by Wu et. al. [Wu21] addresses the incorrect order of error handling code. To identify error handling code this approach however relies on a predefined list of functions that are used in error handling code or values in error checks like ENOMEM. For projects like the Linux kernel that has common error handling functions like panic or checks like IS_ERR, this is a valid approach. However, this approach will not work for students that are implementing their own error handling functions or forget error handling overall.

3 Error Handling

We identify three different classes of error-handling methods: (1) Terminating execution: In case that the error is critical, the error handling terminates the execution. (2) Passing the error upstream and expecting that the caller handles it further. (3) Fixing errors including clean-up and continuing execution.

While error handling is obviously necessary and should be done carefully, the error handling code in the wild is itself sometimes error-prone. Wu et al. [Wu21] report missing memory release, missing refcount release and disordered error handling bugs. The latter occurs for example, when an error handling code does all necessary clean-up operations like memory release, then passes the error upstream, and the caller again does a clean-up (use-after-free bug). Wu et al. found 234 disordered error handling (DiEH) bugs in the Linux kernel v5.3 where most of them have the potential to cause critical security issues. For example, missing memory releases and missing refcount releases may cause memory leaks, which in case of the Linux kernel may crash the whole system (Denial-of-Service). DiEH bugs include for example use-after-free, double-free and NULL-pointer dereference. These bugs may also

Error Category	Number	Error Category	Number
segmentation fault	180	use after free	41
use of uninitialized variable	149	use of unsafe functions	36
memory leak	126	incompatible pointer conversion	21
null dereference	85	double free	19
missing error check for allocation	82	data races	15
buffer overflow	54	bug prone include	12
TCP segmentation violation	52	async signal violation	1
resource leak: non closed streams	47		

Tab. 1: Count of detected errors by CoFee for the operating systems course 2022

lead to security vulnerabilities. For example, CVE-2019-15504[5] and CVE-2019-15292[6] both describe vulnerabilities in the Linux kernel. CVE-2019-15504 is a double-free and CVE-2019-15292 a use-after-free vulnerability. The first is rated critical with a base score of 9.8, the second has base score 4.7 (medium). Further, it is interesting that 87.6 % of these DiEH bugs belong to driver code and not to the kernel itself.

While most of the bugs reported by Wu et al. are meanwhile fixed, the problem of error handling bugs is not solved. See for example CVE-2023-23004[7] from this year with a score of 5.5 (medium), reporting an incorrect error checking bug withing a GPU driver: The code expects a return value to be Null in the error case, whereas it is actually an error pointer. Also security relevant software like OpenSSL contains missing error handling (see for example CVE-2023-0401 NULL dereference during PKCS7 data verification [Moderate severity] from February 2023).

At the University of Potsdam, students learn C during a second semester course and deepen their knowledge within the operating system course (third semester). Since 2021, we are using the CoFee framework for fast continuous feedback for students: The students use Gitlab for code submission. CoFee uses unit tests for functional correctness and runs state-of-the-art code analyzers to enhance the code quality of the submissions. This approach follows the principle of situated learning where the students get familiar with software engineering tools and workflows, which they will face during their professional practice.

The CoFee framework integrates a simple Clang static analyzer (CSA) Plugin, which verifies that the return values of allocation functions are checked for errors. Table 1 shows the count of errors by type detected by CoFee for all student commits in the operating system course in 2022. Compiler errors and unit test failures are not included in the list. However, the table clearly shows that missing error handling (rank 5) is also a problem in student code. And this is just missing error handling for `malloc` and `calloc`.

Figure 1 shows the number of errors during the task progress. The most error handling bugs are reported for the first compilable commit. The figure shows clearly that the students fix

[5] https://nvd.nist.gov/vuln/detail/CVE-2019-15504
[6] https://nvd.nist.gov/vuln/detail/cve-2019-15292
[7] https://nvd.nist.gov/vuln/detail/cve-2023-23004

the reported error handling bugs with each commit, leaving only 3 unfixed bugs in all final submissions. This shows that the feedback helps student to fix these errors. When analyzing the errors by task, they are distributed as follows: 52 errors in the first task, 27 in the second and 3 in the fourth task. This shows that the students learn from their mistakes and improve their code quality in the future.

However since the CSA Plugin only checked for missing error handling for allocation functions, our manual evaluation found 175 other error handling bugs that were not caught. With most of them done in the last task. Since error checking needs to be done for almost all functions in the Glibc, writing unit tests for all functions is unreasonable.

4 CoFee-EHA

To address these shortcomings for novice students, the CoFee Error Handling Analyzer (EHA) leverages that most functions in the Glibc can be categorized into one of the following categories:

1. The function returns a valid pointer if successful, or a NULL pointer (e. g. `malloc`) or -1 (e. g. `mmap`) on failure.
2. The function returns a valid Integer (e. g a file descriptor, see `open`) and -1 on failure.
3. The function returns status information or 0 on success and -1 on failure. The actual results of the function are then returned via an argument pointer (e. g. `read`). Some functions may also return an errno value and 0 on success.
4. The function returns 0 on success and -1 on failure, with no actual result (e. g. `setuid`).

For the first three categories EHA tracks the returned value that indicates if the function was successful or encountered an error (error variable) and the actual result. For the first two categories, this is the same variable. If the result of the function is used, for example in a function call or dereferenced in case of a pointer, while the error variable can still be an error value, a bug is reported. For symbolic execution, the CSA is used.

An example can be seen in Listing 1. The example contains 3 missing error handling bugs. The EHA starts tracking the returned value of `malloc` as error variable and result. When it is loaded as an argument for the `read` function in line 6, EHA checks if `buf` can still be 0, re-using the CSA's symbolic execution engine. The check results in true, since the error handling code in line 4 checks for the wrong value. The missing error handling for `open` is detected analogously. For the `read` call, `res` is tracked as error variable and `buf` as result. When `buf` is accessed at index -1 on line 7, a bug is reported. To detect missing error handling if the result of a function call is not used but returned is handled similarly. Imagine in Listing 1 instead of accessing `buf` in line 7 and 8, `buf` is returned without checking `res` for errors. When `buf` is returned, the symbol is then marked as dead by the CSA. If `res` can still be an error value at this point a bug is reported. To detect missing error handling for the

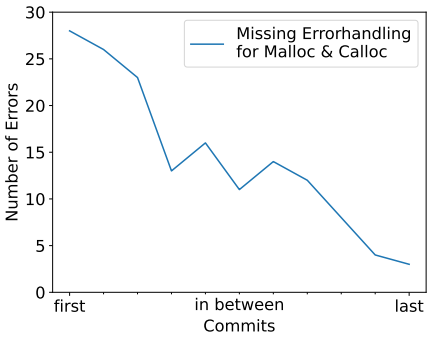


Fig. 1: Count of missing error handling for malloc and calloc per commit

```
1  int main (){
2    int fd;
3    char* buf=malloc(64*sizeof(char));
4    if (buf == (void*)-1 ) {
5      exit(EXIT_FAILURE);
6    }
7    fd = open("/tmp/file", 0);
8    int res = read(fd, buf, 63);
9    buf[res] = '\0';
10   printf("%s\n", buf);
11 }
```

Listing 1: Example for error handling bugs

fourth category, the EHA checks the AST if the returned value of the function is discarded as the UncheckedReturn checker does. It also leverages the CSA to verify that this value is actually further used.

This approach allows detecting error handling bugs for Glibc functions without forcing students to use a certain error handling function or the need to specify the function signature and error codes. If the checker should also detect error handling bugs for non-Glibc functions, the checker allows user specified function specifications for which error handling should also be done. Hence, the checker can easily be tailored to the programming task at hand. An example can be found at our Github repository.[7]

The current EHA however has some limitation. If the result of the function is used in the error handling code, for example to recover from the error, the checker will produce a false positive bug report. To address this limitation, the checker could verify if the result is used inside error handling code. Our evaluation using student code shows however that these false positives are rarely encountered.

There are also a handful of functions that do not indicate errors via return values. strtol is one example. The function has successfully converted the whole string to a long if the input pointer does not point to string terminator, but the pointer pointing to the last converted character does. To our knowledge only type converting functions do not indicate errors directly. In CoFee the correct conversion of input values is covered by unit tests that test correct input validation.

[7] https://github.com/schrc3b6/CoFee-EHA

5 Evaluation

To evaluate the effectiveness of the current state of CoFee-EHA we evaluated the checker on (1) self written tests inspired by the Juliet test suite [Na] and on (2) student submitted code from the operating systems course 2022.

The self written tests consist of 30 tests that are testing functions from all 4 function categories introduced in Section 4, with and without inter-function analysis and using different control structures. Half of these tests include error handling bugs, while the other half are the corresponding correct solutions to detect false positives. CoFee-EHA detected all 15 error handling bugs correctly. However, it reported also 7 false positives. Five of these are caused by freeing a buffer in the error handling code which is currently tracked. The other two are created, because the result symbol is considered dead by the CSA in the error handling code. Both types of false positives can be prevented by not reporting usage or removal of symbols in its own error handling code. The tests and checker are published here.[7]

For student submitted code we used the exercises from the operating systems course of the winter semester 2022. This is the same dataset as referenced in Section 3. It consists of 100 submissions from 8 tasks. All 100 submissions were evaluated manually for error handling bugs. Afterwards CodeChecker[8] was used to analyze the submissions with CoFee-EHA to enable cross translation unit analysis. Out of 178 error handling bugs discovered manually, CoFee-EHA detected 148 bugs, while generating 9 false positive reports. Thus, CoFee-EHA performed with a precision (TP/(TP + FP) of 94.27 % and a recall (TP/ (TP + FN)) of 83.15 %.

6 Conclusion

We presented the error handling analyzer (EHA) for the CoFee framework which aims at teaching secure C programming. EHA utilizes the Clang Static Analyzer to detect error handling bugs. This is supported for the Glibc functions as well as for user specified functions. The latter requires a specification of the function given by the programmer. Our evaluation shows that error handling bugs are relevant and common in student code submissions. We also showed that the feedback generated by CoFee is meaningful for the students and allows them to fix these errors. The results for the current version of the EHA checker showed that it detects 83.15 % of errors with a precision of 94.27 %. We see potential to reduce the number of false positives in future and improve the already good precision.

[8] https://github.com/Ericsson/codechecker

Bibliography

[Be22] Beyer, D.: Progress on Software Verification: SV-COMP 2022. In (Fisman, D.; Rosu, G., eds.): Tools and Algorithms for the Construction and Analysis of Systems. Springer International Publishing, Cham, pp. 375–402, 2022.

[Ch17] Chen, H.-M.; Chen, W.-H.; Hsueh, N.-L.; Lee, C.-C.; Li, C.-H.: ProgEdu - an automatic assessment platform for programming courses. In: 2017 International Conference on Applied System Innovation (ICASI). Pp. 173–176, 2017.

[HK18] Heckman, S.; King, J.: Developing Software Engineering Skills Using Real Tools for Automated Grading. In: Proceedings of the 49th ACM Technical Symposium on Computer Science Education. SIGCSE '18, Association for Computing Machinery, Baltimore, Maryland, USA, pp. 794–799, 2018.

[Ja16] Jana, S.; Kang, Y.; Roth, S.; Ray, B.: Automatically Detecting Error Handling Bugs Using Error Specifications. In: Proceedings of the 25th USENIX Conference on Security Symposium. SEC'16, USENIX Association, Austin, TX, USA, pp. 345–362, 2016.

[KRJ16] Kang, Y.; Ray, B.; Jana, S.: APEx: Automated Inference of Error Specifications for C APIs. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ASE '16, Association for Computing Machinery, Singapore, Singapore, pp. 472–482, 2016.

[KS18] Krusche, S.; Seitz, A.: ArTEMiS: An Automatic Assessment Management System for Interactive Learning. In: Proceedings of the 49th ACM Technical Symposium on Computer Science Education. SIGCSE '18, Association for Computing Machinery, Baltimore, Maryland, USA, pp. 284–289, 2018.

[Na] National Institute of Standards and Technology: Juliet C/C++ 1.3.

[SS22a] Schrötter, M.; Schnor, B.: Leveraging Continuous Feedback in Education to Increase C Code Quality. In: 2022 International Conference on Computational Science and Computational Intelligence (CSCI). CSCI '22, Las Vegas, Nevada, USA, pp. 1950–1956, 2022.

[SS22b] Strickroth, S.; Striewe, M.: Building a Corpus of Task-Based Grading and Feedback Systems for Learning and Teaching Programming. International Journal of Engineering Pedagogy (iJEP) 12/5, pp. 26–41, Nov. 2022.

[TR17] Tian, Y.; Ray, B.: Automatically Diagnosing and Repairing Error Handling Bugs in C. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2017, Association for Computing Machinery, Paderborn, Germany, pp. 752–762, 2017.

[Wu21] Wu, Q.; Pakki, A.; Emamdoost, N.; McCamant, S.; Lu, K.: Understanding and Detecting Disordered Error Handling with Precise Function Pairing. In: 30th USENIX Security Symposium (USENIX Security 21). USENIX Association, pp. 2041–2058, Aug. 2021.

pycheckmate – Addressing Challenges in Automatic Code Evaluation and Feedback Generation for Python Novices

Annabell Brocker[1] and Ulrik Schroeder[1]

Abstract: In academic settings, code assessment differs from traditional software testing by encompassing not only functional correctness but also critical structural aspects like naming conventions and programming concepts. Conventional static analysis tools like Pylint and Flake8, along with input-output unit tests, are deemed inadequate for introductory Python courses. To address this gap, this paper introduces `pycheckmate`, a library, tailored for automatic testing and targeted feedback in introductory Python programming courses.

Keywords: E-Assessment; Programming; Static Analysis; Automated Grading; CS1

1 Introduction

During the assessment process of programming code, automated testing is commonly employed to evaluate the code's static and/or dynamic properties. Statistical analyses are utilized to examine the properties without directly executing the code. Conversely, dynamic analyses involve the execution of the code to assess its functionality, often by comparing input and output [IL20]. The application of automatic assessment tests in introductory courses presents notable challenges, particularly when verifying fundamental aspects such as accurate variable and class naming, as well as the proper use of specific programming concepts [SG14]. Furthermore, code from novices often contains syntax errors, which complicate automated testing. Consequently, lecturers repeatedly create the same checks to initially evaluate the program code accordingly. In the Python programming language, code is dynamically interpreted, so that variables and other instances are not known to the compiler until runtime, whereas in other programming languages, such as Java, they are already known at compile time. Due to these traits, the analysis of specific programming concepts within Python can only be performed if the code has no syntax errors.

This paper therefore presents `pycheckmate`, a library that on the one hand helps lecturers to create automatic tests for introductory Python programming courses and on the other hand allows learners to get more specific feedback on their code (assessment).

2 Related Work

Automatic Assessments Tools The use of automatic assessment of programming tasks is becoming increasingly necessary, as manual corrections are very time-consuming and can

[1] RWTH Aachen University, Learning Technologies Research Group, Ahornstr. 55, 52074 Aachen, Germany
{a.brocker,schroeder}@cs.rwth-aachen.de, https://orcid.org/{0009-0007-6708-0892, 0000-0002-5178-8497}

doi:10.18420/abp2023-2

not be ensured with increasing numbers of novice programmers [Da18]. As a consequence, a wide spectrum of tools has been developed and is being used [SS22]. Reducing it to the programming language Python, still numerous tools are available for the purposes of static and/or dynamic testing, grading, and generation of feedback. An exemplar of an e-assessment system is *JACK*, which not only provides an integrated interface but also enables automated analysis of programming tasks, including Python and Java [Lo15; SG13]. It supports both static and dynamic tests, allowing learners to receive automatically generated feedback upon submission. JACK incorporates dedicated feedback mechanisms, such as variable allocation tables, specifically designed for programming tasks. The university of Turku designed its own web based learning environment called *ViLLE*, which includes task types for programming exercises that can be automatically assessed [LKR18]. Similar to JACK, ViLLE provides learners with immediate feedback upon submission. Additionally, learners have the capability to collaborate on multiple tasks and receive automatically shared feedback on their submissions. *Praktomat*[2], a programming course manager, provides support for the automatic assessment of programming tasks [BHS17]. Lecturers have the ability to define required, optional, and secret checks for submissions. Learners have the flexibility to submit their work at any time, with the required and optional checks being performed initially. If any of the required checks fail, the submission is not accepted. The results of the optional tests are immediately provided to the learner, while the results of the secret tests are revealed after the final accepted submission. The application of the required checks ensures that basic functionalities or requirements within the code are fulfilled, enabling subsequent tests to generate meaningful feedback. Unfortunately, this system lacks support for the assessment of Python programming tasks. Another autograding tool for Python assessments is *nbgrader*[3], which allows instructors to incorporate both public and hidden tests for static and dynamic analyses. It supports various packages, provided they are pre-installed. Learners receive a feedback file containing a comprehensive list of compiler messages, but this format may not be ideal for programming novices.

Static Code Analysis Tools Apart from the automatic assessments tools, Python also has static analysis tools like *Pylint*[4], *Pyflakes*[5] or *flake8*[6]. A comparison conducted by [GP19] evaluates these and other static analysis tools for Python, considering diverse requirements, and provides recommendations accordingly. However, none of these tools are specifically designed for assessing tasks of novice programmers. This limitation arises due to the unique considerations required when examining code from novices, including proper variable, function, and class naming, as well as the correct usage of specific programming constructs [SG14]. *PyTA*[7], for example, is a static code analysis tools built on top of Pylint, but presents

[2] Praktomat, `https://github.com/KITPraktomatTeam/Praktomat/`, Last accessed: 14.06.2023
[3] nbgrader, `https://github.com/jupyter/nbgrader/`
[4] Pylint User Manuel, `https://docs.pylint.org/`, Last accessed: 02.06.2023
[5] Pyflakes GitHub, `https://github.com/PyCQA/pyflakes`, Last accessed: 02.06.2023
[6] Flake8: Your Tool For Style Guide Enforcement, `https://flake8.pycqa.org/en/latest/`, Last accessed: 02.06.2023
[7] PyTA, `https://github.com/pyta-uoft/pyta`, Last accessed: 28.08.2023

error and feedback messages in a manner that is more accessible to novice programmers compared to Pylint itself [LP19]. Currently, PyTA offers more than 100 distinct error and feedback messages, although this count may vary as development continues. However, PyTA is not suitable for assessment purposes due to challenges in configuration for lecturers and the absence of important checks required for automatic assessment, such as verifying correct parameter naming. Instead, PyTA focuses on providing valuable feedback to novice programmers within the context of conventional development processes. A further analysis tool for Python is *Scalpel*[8], which includes several features such as the construction of a control-flow graph and the representation of static single assignment [LWQ22]. This library proves to be particularly valuable for visualizing step-by-step code execution for novice programmers. Consequently, Scalpel not primarily intended for automatic evaluation, but rather serves as a means to automatically generate more detailed and comprehensive feedback. *ASPA*[9] is primarily dedicated to analyzing programming code generated by novice programmers in order to provide feedback [Lu22]. It offers a graphical user interface that facilitates the insertion of code files for examination. A study was conducted to evaluate the usefulness of ASPA, among other aspects. The findings revealed that ASPA significantly assisted the majority of novice programmers with their weekly exercises. However, the investigations conducted with ASPA were solely general in nature and did not include contextual investigations. Regrettably, the documentation available for this tool is insufficient, preventing a definitive conclusion on its effectiveness from being drawn at this stage.

Abstract Syntax Tree An Abstract Sytnax Tree (AST) serves as a hierarchical representation of program source code in the form of a tree structure. The Python programming language provides a built-in module called *ast*[10], which allows the conversion of Python code into a valid AST. By examining the transformed code within the AST, specific properties can be verified, such as the presence of particular names or the usage of specific libraries. For instance, Fig. 1 illustrates the AST derived from the one-liner `a = max (1, 2, 3)`. To determine whether the learner has implemented a variable named `a`, one can traverse the individual nodes of the tree until the corresponding nodes are found. In this case, it would be via `source_tree.body[0].targets.id` if the code has been previously parsed and stored in the `source_tree` variable. Unfortunately, only Python code devoid of SyntaxErrors can be successfully converted into a corresponding AST using the built-in ast module. Numerous existing tools, including Pylint, Pyflakes, and ASPA, rely on the conversion of code into an AST for their functionality.

Feedback In the domain of learning and teaching, feedback is an indispensable element for enhancing effectiveness [HT07]. This also applies in the area of automated assessment,

[8] Scalpel: The Python Static Analysis Framework, `https://github.com/SMAT-Lab/Scalpel`, Last accessed: 02.06.2023

[9] ASPA - Abstrakti SyntaksiPuu Analysaattori, `https://github.com/RoopeLuukkainen/ASPA`, Last accessed: 02.06.2023

[10] ast — Abstract Syntax Trees, `https://docs.python.org/3/library/ast.html`, Last accessed: 12.05.2023.

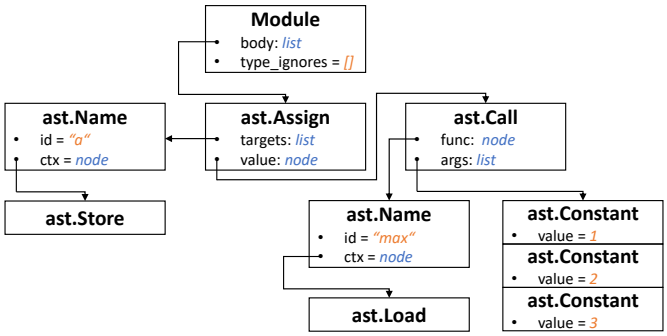


Fig. 1: Abstract Syntax Tree for Python code: `a = max (1, 2, 3)`

where feedback is generated automatically. Especially for programming novices, the display of an extensive callstack of compiler messages often proves to be unhelpful, as it can lead to cognitive overload [WLG12]. Additionally, novice programmers benefit from feedback that recognizes even the smallest accomplishments, providing positive reinforcement alongside identification of errors [MAG18]. Within programming courses, feedback can be classified into various types [KJH18; Na08]. Multiple learning environments, like previous presented tool JACK, have been investigated in terms of their feedback they provide and, based on this, developed guidelines on when and how to intervene [Je22]. Extensive research has demonstrated that elaborative feedback, such as compiler messages or their customization specifically for novice learners, greatly contributes to the learning process [Ha21].

3 pycheckmate

`pycheckmate` is a library designed to analyze Python code for various properties, focusing primarily on the evaluation of code authored by programming novices. A first prototype was developed as part of a bachelor thesis [St23]. Unlike other statistical analysis tools, `pycheckmate` does not primarily address code styling; instead, it enables the examination of SyntaxErrors, the usage of programming concepts, and the correct designation of variables, functions, classes, and other entities. All analysis are based on the AST of a Python program code. Through interviews with tutors who supervised programming lectures for Python programming novices, the most relevant aspects to investigate were identified and summarized in Tab. 1. To utilize the library, the code to be examined must be transformed once into an AST by initialising a `pycheckmate` object. This approach optimizes execution time and memory usage. Subsequently, each test can be invoked on the object instance, returning a dictionary indicating the success (`bool`) of the test and a corresponding feedback message (`str`). Intentionally, no exceptions are thrown to prevent the termination of subsequent tests, unless significant configuration is required. Within the scope of `pycheckmate`, the examination of code is limited to its current state. For instance, the analysis of indirect recursion within a function is contingent upon the presence of the

relevant functions within the code itself. `pycheckmate` is an actively evolving tool that is openly accessible to users[11].

Requirement	Explanation
Code Compilation	Check if the code has SyntaxErrors or not.
Variable Name	Check if a specific variable is defined (inside a function).
Function Name	Check if a specific function or multiple function are defined.
Class Name	Check if a specific class is defined.
Function / Class Parameters	Check if a function / class has specific parameters including correct naming, correct type as well as default value.
Module Import	Check if a specific module or a specific function from a module has been imported.
Module Usage	Check if specific function from a module has been used.
Function Usage	Check if specific function has been used.
Programming Concept Usage	Check if the the following programming concepts have been used: for including iteration range, while, lambda, comprehension, try-except.
Recursive Function	Check if a function is recursive (direct and indirect).
File Handling	Check if a file was opened, closed or both via with.

Tab. 1: Results from requirements evaluation.

Feedback Generation To cater to the needs of novice programmers, in line with the recommendations of [MAG18], the library provides static checks for even the most basic and successful programming activities. These checks include feedback messages specifically tailored for novice learners. For example, the function $function\_has\_parameters$, which configuration can be found in List. 1, informs novices which necessary named parameters are still missing or, if the parameters are available, whether the type and the default value are also correct:

Parameter 'param1' of function 'test_function' has wrong default.
Expected default: '0', got '1'.

Feedback messages are also generated for passed tests, and not only for failed tests to honour successful actions. When certain checks rely on the presence of others, for instance, a specific function's existence, such interdependencies are communicated to the learner as well. The messages are currently provided in English, but should also support multilingualism in the future. If the pre-configured feedback messages are not sufficient, they can be overwritten or extended by the lecturers. The timing of feedback message delivery to learners is presently determined by the lecturer due to the underlying architecture.

Performance Measures `pycheckmate`'s performance was rigorously assessed through extensive testing, measuring execution time and memory usage. Execution time was recorded as the duration between the start and end of each check, and memory consumption was

[11] pycheckmate, `https://doi.org/10.17605/OSF.IO/BR68W`, last accessed 28.08.2023

analyzed using the `tracemalloc` module. We conducted 1000 iterations on various devices, including a Windows 10 Pro laptop (i7-10510U, 16 GB RAM) and a Raspberry Pi 4 (4 GB RAM) running Raspberry Pi OS. Results showed execution times were device-dependent, as detailed in Table 2. Nevertheless, `pycheckmate` performed well on typical devices encountered in practice.

	Laptop	Raspberry Pi 4
Execution Time Average	~2.5 ms	~11.2 ms
Memory Usage Average	~0.15 MB	~0.05 MB

Tab. 2: Performance measurements of `pycheckmate`.

Use Cases The presented library exhibits versatile applicability in various scenarios. Firstly, it can be traditionally employed for evaluating programming tasks, enabling lecturers to seamlessly integrate the library into their existing frameworks and leverage the pre-defined tests. This integration offers lecturers enhanced accessibility and efficiency in automating the assessment of Python code. Additionally, the library finds utility in both summative and formative e-examinations, where automatic evaluation mechanisms are employed for subsequent analysis. One notable challenge in utilizing automatic evaluation mechanisms in e-examinations, where learners are restricted to using basic text editors without execution capabilities, lies in the inability to automatically evaluate the code itself due to syntax errors or incorrect variable and class naming. To address this, the library can be employed to verify specific code specifications, allowing students to receive feedback on those aspects, similar to the required checks implemented in the Praktomat programming course manager. However, incorporating student executability would entail certain trade-offs, such as rendering certain tasks, like assessing code reading comprehension, unfeasible within the examination context. For e-examination systems conducted through web applications, the library can be integrated using Pyodide[12], provided that appropriate interfaces are established beforehand within the e-examination system. An example to integrate `pycheckmate` can be found in List. 1.

```
from pycheckmate import PyCheckMate

#store code as str in variable source_code
with open("testing_file.py") as file:
    source_code = file.read()

reqs_args = {
    'param1': { 'default': 0 },
    'param2': { 'type': int }
}

pcm = PyCheckMate(source_code)
```

[12] Pyodide, `https://pyodide.org/en/stable/`, Last accessed: 26.05.2023

```
check_func_name = pcm.has_function("test_function")
check_func_params = pcm.function_has_parameters("test_function",
    required_args=reqs_args, required_vararg=False, required_kwarg="
    kwarg")
```

List. 1: Example configuration to integrate `pycheckmate` into an automatic assessment tool.

4 Conclusion and Future Work

In this paper we discussed problems of automatic assessment for introductory programming courses and presented the library `pycheckmate` which is designed to analyze Python code accordingly. The library not only checks for the existence of basic programming concepts, but also generates feedback suited to novices, which can, however, be adapted by the lecturer according to the context. Up to now, the feedback messages have not yet been evaluated with the actual user group (programming novices). This must be done retrospectively in order to also prove the positive effect of the messages adapted to the novices. As part of a requirements analysis, fundamental features were identified and implemented. Nevertheless, the library is in continuous development, so that further features will be added in the future. Furthermore, it might be possible to integrate code styling checks based on existing static analysis tools, bundling the most important code styling checks for programming novices in one library.

Bibliography

[BHS17] Breitner, J.; Hecker, M.; Snelting, G.: Der Grader Praktomat. In: Automatisierte Bewertung in der Programmierausbildung. Digitale Medien in der Hochschullehre 6, Waxmann Verlag GmbH, pp. 159–172, 2017, URL: https://www.waxmann.com/automatisiertebewertung/.

[Da18] Dawson, J. Q. et al.: Designing an Introductory Programming Course to Improve Non-Majors' Experiences. Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE)/, pp. 26–31, 2018.

[GP19] Gulabovska, H.; Porkoláb, Z.: Survey on Static Analysis Tools of Python Programs. In: Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA). 2019.

[Ha21] Hao, Q. et al.: Towards understanding the effective design of automated formative feedback for programming assignments. Computer Science Education 32/, pp. 105–127, 2021.

[HT07] Hattie, J. A. C.; Timperley, H. S.: The Power of Feedback. Review of Educational Research 77/, pp. 112–81, 2007.

[IL20] Ismail, M. H.; Lakulu, M. M.: A Critical Review on Recent Proposed Automated Programming Assessment Tool. Psychology and Education/, pp. 1049–1060, 2020.

[Je22] Jeuring, J. et al.: Towards Giving Timely Formative Feedback and Hints to Novice Programmers. Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education/, 2022, URL: https://api.semanticscholar.org/CorpusID:255226835.

[KJH18] Keuning, H.; Jeuring, J.; Heeren, B.: A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. ACM Transactions on Computing Education (TOCE) 19/, pp. 1–43, 2018.

[LKR18] Laakso, M.; Kaila, E.; Rajala, T.: ViLLE – collaborative education tool: Designing and utilizing an exercise-based learning environment. Education and Information Technologies 23/, pp. 1655–1676, 2018.

[Lo15] Lohmann, E.: Erweiterung eines E-Assessment-Systems um eine Prüfkomponente für die Programmiersprache Python. In: Proceedings of the Second Workshop "Automatische Bewertung von Programmieraufgaben". 2015.

[LP19] Liu, D.; Petersen, A.: Static Analyses in Python Programming Courses. In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE). Pp. 666–671, 2019.

[Lu22] Luukkainen, R. et al.: ASPA: A Static Analyser to Support Learning and Continuous Feedback on Programming Courses. An Empirical Validation. 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)/, pp. 29–39, 2022.

[LWQ22] Li, L.; Wang, J.; Quan, H.: Scalpel: The Python Static Analysis Framework. ArXiv abs/2202.11840/, 2022.

[MAG18] Mattheiss, S. R.; Alexander, E. J.; Graves, W. W.: Elaborative feedback: Engaging reward and task-relevant brain regions promotes learning in pseudoword reading aloud. Cognitive, Affective, & Behavioral Neuroscience (CABN) 18/, pp. 68–87, 2018.

[Na08] Narciss, S.: Feedback Strategies for Interactive Learning Tasks. In: Handbook of Research on Educational Communications and Technology (3rd ed.) Pp. 125–144, 2008.

[SG13] Striewe, M.; Goedicke, M.: JACK Revisited: Scaling Up in Multiple Dimensions. In: European Conference on Technology Enhanced Learning (ECTEL). Pp. 635–636, 2013.

[SG14] Striewe, M.; Goedicke, M.: A Review of Static Analysis Approaches for Programming Exercises. In: Computer Assisted Assessment. Research into E-Assessment (CAA). Pp. 100–113, 2014.

[SS22] Strickroth, S.; Striewe, M.: Building a Corpus of Task-Based Grading and Feedback Systems for Learning and Teaching Programming. Int. J. Eng. Pedagog. 12/, pp. 26–41, 2022.

[St23] Stuermer, M.: Development of a library to analyze Python code used in ACE editor, Bachelor's Thesis, RWTH Aachen University, 2023, URL: https://doi.org/10.18154/RWTH-2023-03637.

[WLG12] Watson, C.; Li, F. W. B.; Godwin, J. L.: BlueFix: Using Crowd-Sourced Feedback to Support Programming Students in Error Diagnosis and Repair. In: International Conference on Advances in Web-Based Learning (ICWL). 2012.

Automatic Evaluation of Haskell Assignments Using Existing Haskell Tooling

Thomas Prokosch[1] and Sven Strickroth[2]

Abstract: Learning Haskell is hard for many students because of its functional nature. What is more, Haskell uses a sophisticated type system that many students find quite confusing in the beginning. Therefore, providing timely feedback regarding correctness and quality for student Haskell solutions is important, a challenge particularly in large courses. Computer-assisted correction of submissions offers a way to help tutors and students alike, but this requires the computer to understand the language. Parsing the student submissions into a syntax tree and analyzing the syntax tree is one possibility, however, this paper describes a more straightforward approach that uses only a Haskell compiler and a few standard tools. Based on a migration of a Haskell course with manual to automatic correction we classified assignment into different categories and describe this approach in detail for each category.

Keywords: Haskell; automatic evaluation; e-assessment; programming education

1 Motivation

The programming language Haskell is used in computer science education due to its conciseness, its strong static typing, and as an example for a functional programming language. For these reasons the language is also said to be hard to learn, and students not familiar with functional programming may have a hard time to grasp its concepts [TUI15]. Helping students to learn the language and assessing their understanding of it can be challenging, especially in large classes. The manual evaluation of submissions by teaching assistants, however, is time-consuming and can be prone to errors. Given these circumstances, providing timely feedback to students is hard, hindering the learning process and limiting the potential for improvement.

The experiences described here are based on the second semester course “Programming and Modelling with Haskell” at LMU Munich, Germany held in summer term of 2022 which has been attended by several hundreds of students. This course was migrated from manually corrected assignments to an automated approach using GATE to support tutors providing feedback on the voluntary weekly homework assignments and to allow students to request a first timely feedback from the system on their submissions.

The research question is how can the Haskell compiler and interpreter be used for checking correctness and providing feedback to tutors and students without the need to manually

[1] LMU Munich, Institut für Informatik, Oettingenstr. 67, 80538 München, thomas.prokosch@ifi.lmu.de
[2] LMU Munich, Institut für Informatik, Oettingenstr. 67, 80538 München, sven.strickroth@ifi.lmu.de, https://orcid.org/0000-0002-9647-300X

doi:10.18420/abp2023-3

parse the submissions. Different types of program evaluation are identified based on the existing assignments of the said course and described exemplarily.

2 Related Research

Based on the corpus on task-based e-assessment-systems by Strickroth & Striewe [SS22], only 15 of the 178 analyzed tools support Haskell, of which six systems do not provide any further information to support this claim. Tools particularly designed for Haskell are the intelligent tutoring system *Ask-Elle* [GJH12] and *hsExprTest* [Št20]. Also, there exists *AutoTool* which provides automatic assignment generation for Haskell assignments [We21].

For testing Haskell programs, *QuickCheck* is often being used as it is able to automatically generate test cases based on a provided specification (e. g., [Be18; BHS17; SVW19]). However, QuickCheck only generates random test cases covered by the specification which is why bugs may be missed for two reasons: First, they can be missed due to randomness and the limited sample size (its developers acknowledge this fact, cf. [Be18]). Second, the test case specification may be faulty just as any program code can be faulty. Either way, QuickCheck alone cannot be used with effectful programs. Therefore, some systems use a combination of QuickCheck and *HUnit* (e. g., Praktomat [BHS17]) or combine different testing techniques such as *HLint*, QuickCheck, *IOSpec*, etc. (e. g., AutoTool [SVW19]) to improve the assessment and feedback. Simple I/O tests also seem to be used in *flop* [LMP12].

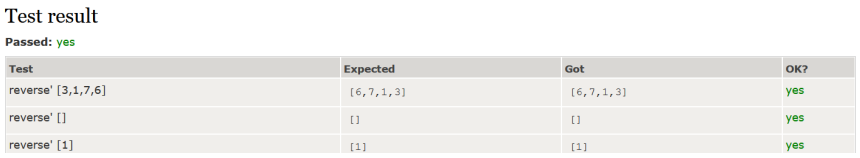
In summary, it can be said that — with few exceptions for very specific use cases such as for Ask-Elle — there are no descriptions on how code tests are implemented in detail and there are no publications known to the authors that assess Haskell type definitions without parsing the Haskell program into an abstract syntax tree (AST). The goal of this work is to outline what can be checked – beyond simple I/O blackbox tests – using existing Haskell compilers and interpreters, and tools.

3 Checking Student Submissions: Patterns for Program Evaluation

Due to the complexity of Haskell and the variety of assignments and their associated learning objectives, program evaluation requires the combination of different tools. The tools under investigation are the interpreter `runghc`, the compiler `ghc`, the read-eval-print-loop `ghci` and various tools from the Linux operating system such as `grep`, `cat`, and shell built-ins. After implementing the first tests to check student submissions, a few implementation patterns emerged. In particular, test code can be classified into four different types, all of which have been implemented in GATE, that are described in more detail below: First, testing effectful Haskell programs just as any other black-box executable would be tested. For this reason, this test type will be described only briefly, still outlining some common issues. Second, testing pure Haskell functions and simple custom data types; third, testing complex custom data types; and fourth, testing custom modules.

3.1 Implementation of the Haskell Test Driver and Black-Box-Testing

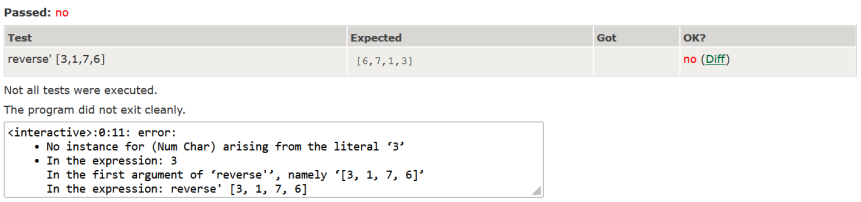
The Haskell tests are implemented as I/O-based tests in GATE using (Docker) container technology for secure execution: For each test, a new container is created based on a pre-defined image that primarily contains the Bourne Again Shell `bash` and the Glasgow Haskell Compiler `ghc`. Each container is read-only, except for the temporary directory `/tmp` and the `/home` directory to which the test driver and student submission are mapped, has limited capabilities, disabled swap, a memory limit, disabled networking, is limited to one CPU core, and has a hard ulimit for files. The test driver script is executed using an unprivileged user and will be terminated after a specified amount of time[3].


Test result

Passed: yes

Test	Expected	Got	OK?
reverse' [3,1,7,6]	[6,7,1,3]	[6,7,1,3]	yes
reverse' []	[]	[]	yes
reverse' [1]	[1]	[1]	yes

Fig. 1: Example of how the test results are reported to students and tutors


Passed: no

Test	Expected	Got	OK?
reverse' [3,1,7,6]	[6,7,1,3]		no (Diff)

Not all tests were executed.

The program did not exit cleanly.

```
<interactive>:0:11: error:
    • No instance for (Num Char) arising from the literal ‘3’
    • In the expression: 3
      In the first argument of ‘reverse’’, namely ‘[3, 1, 7, 6]’
      In the expression: reverse' [3, 1, 7, 6]
```

Fig. 2: Example of how a failed test is reported to students and tutors

The test driver is based on a shell script that is automatically generated by GATE. This script consists of preparation/compilation code and the code of an arbitrary number of test steps. For each test step, a title, some shell code for the test as well as the expected output are required as inputs in GATE. The test driver is implemented as simple as possible: When the test is executed, the test driver first executes the test's preparation code. Here, also common functions can be defined for reuse from the individual test steps. If this preparation code fails, failure is reported as a syntax error in GATE. If it succeeds, a separator line is printed to STDOUT and STDERR to distinguish its output from the outputs of the test steps to come. Subsequently, all test steps are executed in a similar manner until one of the steps fails or all steps have been executed successfully. Next, GATE splits the output according to the separator lines and the individual chunks of output are compared to the expected output. Regardless of the outcome, the actual and the expected output is presented to the students and/or tutors (cf. Fig. 1). The test step titles indicate to the students the functions and arguments which were tested so that the test itself is comprehensible. In case of an error, the output on STDERR after the last separator line is presented to the students additionally to the STDOUT results (if not disabled). If there is any error or attempt to prematurely end the tests, GATE can detect this by comparing the number of separator lines with the

[3] link to GATE implementation: `https://github.com/csware/si`

expected number of test steps. Figure 2 shows an example of an error detected by Haskell that occurred after the preparation step.

Even if black-box tests are quite common, there are two aspects to consider: The locale and encoding needs to be configured (`export LC_ALL="C.UTF-8"`): First, to get consistent results as many tools produce different output for different locales. Second, to prevent errors such as `<stdout>: commitBuffer: invalid argument (invalid character)` which arise because Haskell checks the encoding of output characters based on the configured locale when students use umlauts in their code (e. g., `putStrLn "Übersetzung:"`). There are also several ways to output text on the console. For example, the Haskell function `print` includes quotation marks around strings while the `putStrLn` does not. Mixing the two functions or varying the output slightly from the specification significantly complicates I/O tests. Hence, the tests need to be constructed in a way that they check for the wanted output (e. g. using `grep` and/or regular expressions) and also for output that needs to be absent (e. g., in a translator program, words which are not the correct translations must not be printed). If the check is successful, the test step prints out the hard-coded "expected" text to the console, and the received (incorrect) result otherwise. See the following code as an example:

```
result=$(echo "gut" | ./translator)
[[ $? != 0 ]] && exit 1
set +e
echo $result | grep -q "guad"; case1=$?
echo $result | grep -q "servus"; case2=$?
set -e
if [[ $case1 -eq 0 ]] && [[ $case2 -eq 1 ]]; then
    echo "guad"
else
    echo $result
fi
```

3.2 Checking pure Haskell programs and simple custom data types

Checking pure Haskell functions is much easier than checking effectful programs. More often than not, no preparation script is necessary, and the test steps can simply rely upon the functionality of GHCi. This approach offers two advantages: For submissions with no syntax errors, built-in standard equality functionality can be used to compare the actual result with the expected result. For submissions with syntax errors, the Haskell interpreter GHCi returns an error message and a non-zero exit code which is then picked up by GATE and then shown to the user.

```
ghci -e "pi_approx 6 < 3.0 && pi_approx 6 > 2.99" 4-2a.hs
```

The above code shows an example for a script checking a pure Haskell function. The student's code is expected to reside in the file `4-2a.hs` while the test code is given as an expression on the command line. Due to the use of GHCi, code is only interpreted, not compiled. This increases execution speed because no binary files need to be written to disk only to be read right thereafter. The students were required to write a pure function `pi_approx :: Int -> Double` with well-defined behaviour. The `Double` valued result of the function is checked in the standard numeric way returning `True` or `False`, depending on the student's implementation.

In a similar manner, the definition of simple data types can be checked. One assignment asked students to specify a data type to represent playing cards. No functions were required to be implemented on that data type (in particular, deriving the type classes `Show`, `Enum`, or `Eq` were not part of this exercise) but the arguments to the data constructor have been specified and need to be checked. This was done by constructing a value and assigning that value to a variable, as is shown below. The variable assignment may seem redundant but construction without variable assignment would trigger a warning with some GHC versions. It should also be pointed out that the first line in the example does not produce any output in case of success, since an assignment in Haskell must not have any side effects. The second line shows how an output for students can be generated. In the error case GHCi produces an error message which will then be shown to the user for both variants.

```
ghci -e "card = Card Club  Two" 7-1.hs
ghci -e "card = Card Heart Two" 7-1.hs && echo "Heart correct"
```

3.3 Checking the application of pure functions on complex custom data types

For some assignments the students were expected to define their own data types with a given name and given constructors but without explicitly requiring them to derive `Show` or `Eq` type classes. However, verifying values of these types is much easier with these type classes, so deriving them stand-alone (outside of the student's code) is necessary. For this, the GHC extensions `StandaloneDeriving` (and sometimes also `FlexibleInstances`) were used to augment the students' code.

```
echo "
:set -XStandaloneDeriving
:set -XFlexibleInstances
deriving instance {-# OVERLAPPING #-} Eq (ML Char)

L '4' E == myTail (L '3' (L '4' E))
" | ghci -v0 7-2.hs
```

The above code shows an example for a stand-alone derivation of the `Eq` type class. First, both extensions of GHCi are being enabled; the extension `FlexibleInstances` is required in this example as the data type `ML a` is a parametrized data type. The `OVERLAPPING` directive makes sure that the code still compiles when the student also provides an implementation of these type classes. Following these definitions, the data type itself is checked by applying the student-provided function `myTail` to some input values utilizing the constructors `L` and `E` and comparing the result with the expected result. This example also shows that, even with complex code like the one above, feedback can be generated, as the output of the test step is simply `True` or `False`, indicating the correctness of the solution.

The command line option `-v0` of `ghci` disables verbosity of GHCi which is required to not pollute the students' output with GHC messages regarding the redefinition of the type class(es). Furthermore, supplying the teacher's code as an expression (by the use of the command line option `-e`, not as part of a input file) is necessary to not modify the student's submission. As a result, line numbers do not change and students exactly know the location of a possible error. In general, this approach is helpful when the teacher needs to supply functions not defined in a type class. One particular use case is the conversion of a student-defined data type to a standard Haskell data type which comes in handy when the user was expected to implement type classes `Show` and `Eq` which must be checked but not be trusted to be correct. In this case, the student-defined data type is converted to a standard Haskell data type that is using the standard (and thus correct) `Eq` and `Show` type classes.

In the following example, the function `mlToList` (defined by the lecturer) is added to convert a binary tree `ML a` to a list representation after a student-supplied function `myAdd` operated on two binary trees. The expected outcome of the student's code is again a binary tree which is then converted to a standard Haskell list. This list is then compared to the example solution, yielding `True` if the student provided a correct implementation and `True` otherwise.

```
echo "
:{
  mlToList E = []
  mlToList (L head tail) = head:(mlToList tail)
:}

mlToList (myAdd (L 1 (L 2 (L 3 (L 4 E)))) E) == []
" | ghci -v0 7-2.hs
```

3.4 Checking custom modules

Student-defined modules can also be tested by augmenting the students' code with GHCi specific commands, as seen in the following preparation code:

```
preproc () {
echo -e '
:m
import Bank
'"$1"
}
run () {
    local prog=$(preproc "$1")
    echo "$prog" | ghci -v0 12-1.hs
}
```

The shell function `preproc` prepares test code to run in the context of the student's module by using the GHCi command `:m` in order to revert the loaded modules to a predefined state. Subsequently, the `import` statement brings the students' module into scope. The shell function `run` is a template to actually run the student code; it needs to be instantiated by giving a Haskell expression as a parameter, such as `run "deposit 30 (-70, 300) == Right (-70,330)"`.

4 Discussion, Conclusions & Outlook

Based on a migration from a "traditional" Haskell lecture with manually corrected assignments this paper investigated how the submissions can be automatically evaluated mainly by using existing Haskell tooling. Four different assignment types such as testing pure Haskell functions and simple as well as complex custom data types were identified and described in more detail. The results indicate that a variety of tests can be performed this way without the need of additional libraries, parsers, or sophisticated frameworks. This might also offer new ways for teaching approaches that use CI/CD techniques. Still, using such additional tools can be helpful to provide a more holistic/detailed feedback/assessment as more details on incorrect solution attempts can be provided. – Note that even very simple correct/incorrect feedback has shown to be helpful for students and tutors [SOP11]. – Additionally, as this approach fully relies on GHC tooling, unclear error messages still might be a problem [TUI15] that needs to be further investigated and feedback messages to be improved or explained to students (cf. [Je22]). For example, common errors could be collected and then specific feedback/explanations provided for those errors later (e. g., detection using *grep*). In the description we mentioned that by using GHCi a separate compilation step can be saved. However, a separate compilation step can still make sense regarding the test results which are presented to the students. Then, a syntactically invalid code is not associated with the first test step.

A major effort was to design and adjust the assignments for the automatic evaluation. Prior to the switch to GATE, most assignments were designed with quite some freedom for the students. For example, file names and the concrete input and output formats were hardly specified. This gives more freedom to the students, however, makes automatic testing close

to impossible. Writing good tests is not easy and requires experience. Also, newly developed tests need to be monitored as the "problem" can also be outside of the test code in the environment: Having not configured the encoding led to a significant number of false negatives.

A preliminary evaluation with randomly sampled final versions of the students' submissions showed that all test types can tell erroneous and correct submissions apart. The test output was also provided to tutors as an aid for the correction. The tutor feedback, however, seem to not always mention found issue(s) by the tests.

A detailed evaluation how the students used the feedback, how helpful it was, and whether students were able to improve their solution attempt upon it, is planned and in preparation.

Bibliography

[Be18] Benotti, L.; Aloi, F.; Bulgarelli, F.; Gomez, M. J.: The Effect of a Web-based Coding Tool with Automatic Feedback on Students' Performance and Perceptions. In: Proc. SIGCSE TS. Pp. 2–7, 2018.

[BHS17] Breitner, J.; Hecker, M.; Snelting, G.: Der Grader Praktomat. In: Automatisierte Bewertung in der Programmierausbildung. Waxmann, pp. 159–172, 2017.

[GJH12] Gerdes, A.; Jeuring, J.; Heeren, B.: An interactive functional programming tutor. In: Proc. ITiCSE. Pp. 250–255, 2012.

[Je22] Jeuring, J.; Keuning, H.; Marwan, S.; Bouvier, D.; Izu, C.; Kiesler, N.; Lehtinen, T.; Lohr, D.; Peterson, A.; Sarsa, S.: Towards Giving Timely Formative Feedback and Hints to Novice Programmers. In: Proc. WG Reports ITiCSE. ACM, Dec. 2022.

[LMP12] Llana, L.; Martin-Martin, E.; Pareja-Flores, C.: FLOP, a free laboratory of programming. In: Proc. Koli Calling. Pp. 93–99, 2012.

[SOP11] Strickroth, S.; Olivier, H.; Pinkwart, N.: Das GATE-System: Qualitätssteigerung durch Selbsttests für Studenten bei der Onlineabgabe von Übungsaufgaben? In: Proc. DeLFI. Pp. 115–126, 2011, URL: https://dl.gi.de/handle/20.500.12116/4740.

[SS22] Strickroth, S.; Striewe, M.: Building a Corpus of Task-based Grading and Feedback Systems for Learning and Teaching Programming. iJEP 12/5, pp. 26–41, 2022.

[Št20] Štill, V.: Automatic Test Generation for Haskell Programming Assignments. In: Proc. ITiCSE. 2020.

[SVW19] Siegburg, M.; Voigtländer, J.; Westphal, O.: Automatische Bewertung von Haskell-Programmieraufgaben. In: Proc. ABP. 2019.

[TUI15] Tirronen, V.; Uusi-Mäkelä, S.; Isomöttönen, V.: Understanding beginners' mistakes with Haskell. Journal of Functional Programming 25/, e11, 2015.

[We21] Westphal, O.: A Framework for Generating Diverse Haskell-I/O Exercise Tasks. In: Functional and Constraint Logic Programming. Springer, pp. 97–114, 2021.

Wieviel Ähnlichkeit ist in Programmierprüfungen normal?

Christoph Olbricht [1], Rafael Schypula[2] und Michael Striewe [3]

Abstract: Werkzeuge zur Identifikation von Plagiaten in Programmieraufgabe messen häufig die Ähnlichkeit zwischen Abgaben. Die Interpretation dieser Werte zur Identifikation von Verdachtsmomenten ist jedoch schwierig, da eine gewisse Ähnlichkeit von Abgaben unvermeidlich ist. Es ist insbesondere unklar, welche Ähnlichkeitsverteilung innerhalb einer Kohorte ohne Plagiate als normal angesehen werden kann und welche Abweichungen davon als Verdachtsmoment genutzt werden können. Der vorliegende Beitrag vergleicht dazu unter Nutzung zweier Werkzeuge zur Plagiatserkennung unter Aufsicht erstellte Abgaben zu Prüfungsaufgaben mit solchen, die ohne Aufsicht entstanden sind. Die Ergebnisse zeigen, dass die Ermittlung einer „Baseline“ für die erwartbare Ähnlichkeit schwierig ist, da auch schon kleine Änderungen der Aufgabenstellung starke Auswirkungen auf die Ähnlichkeit der Lösungen haben können.

Keywords: Programmieraufgaben, Plagiate, Programmähnlichkeit, Täuschungsversuche, JPlag, Sherlock

1 Einleitung

Plagiate bei der Lösung von Programmieraufgaben sind ein lange bekanntes Problem in der Lehre, zu dessen Bewältigung zahlreiche Werkzeuge zur Erkennung von Plagiaten entwickelt wurden. Diese Werkzeuge erkennen Plagiate nicht direkt, sondern messen die Ähnlichkeit zweier Lösungen. Es bleibt letztlich Aufgabe der Lehrenden zu entscheiden, ob es sich dabei um ein Plagiat handelt [NJK19]. Ein Stolperstein ist dabei, dass Programmierung ein strukturiertes Vorgehen bei der Entwicklung einer Lösung erfordert. Folgen Studierende unabhängig voneinander demselben – beispielsweise in der Lehre vermittelten – Vorgehen, so weisen auch ihre Lösungen mit einer gewissen Wahrscheinlichkeit eine hohe Ähnlichkeit auf. Bei der Verfolgung eines Einzelfalls muss daher immer mit einer rein zufälligen Ähnlichkeit gerechnet werden. Treten jedoch in einer Menge von Lösungen deutlich mehr Fälle von hoher oder sehr hoher Ähnlichkeit auf, als im Schnitt zu erwarten sind, ist dies ein Indikator dafür, dass nicht alle diese Fälle rein zufällig entstanden sind. Unabhängig von der verwendeten Methode eines Werkzeugs zur Plagiatserkennung und dessen Trefferquote beim Aufspüren ähnlicher Lösungen ist es daher für

[1] Universität Duisburg-Essen, paluno – The Ruhr Institute for Software Technology, Gerlingstraße 16, 45127 Essen, christoph.olbricht@paluno.uni-due.de, https://orcid.org/ 0009-0002-4823-7894
[2] Universität Duisburg-Essen, paluno – The Ruhr Institute for Software Technology, Gerlingstraße 16, 45127 Essen, rafael.schypula@paluno.uni-due.de
[3] Universität Duisburg-Essen, paluno – The Ruhr Institute for Software Technology, Gerlingstraße 16, 45127 Essen, michael.striewe@paluno.uni-due.de, https://orcid.org/0000-0001-8866-6971

doi:10.18420/abp2023-4

Lehrende wichtig zu wissen, mit welcher Wahrscheinlichkeit sie in einer Menge von Lösungen mit Fällen von hoher oder sehr hoher Übereinstimmung rechnen müssen.

Der vorliegende Beitrag vergleicht dazu konkret die Ergebnisse von zwei Werkzeugen zur Plagiatserkennung auf Lösungen von Aufgaben, die unter Aufsicht bearbeitet wurden und die somit keine Plagiate enthaltenen können, mit Ergebnissen auf Lösungen vergleichbarer Aufgaben, die nicht unter Aufsicht erstellt wurden und die somit Plagiate enthalten können. Damit soll herausgefunden werden, ob es eine nachweisbare Änderung in der Ähnlichkeitsverteilung gibt, die als Verdachtsmoment verwendet werden kann.

1.1 Verwandte Arbeiten

Strickroth [St21] vergleicht Werkzeuge zur Plagiatserkennung und verwendet dazu reale Daten, bei denen teilweise die Zahl der enthaltenen Plagiate bekannt ist. Die Untersuchung fokussiert die Übereinstimmung zwischen den verglichenen Werkzeugen und die Rate bei der Erkennung der Plagiate. Sie liefert keine Aussagen darüber, welche Ähnlichkeitswerte in Lösungen normal sind und bezeichnet den verwendeten Grenzwert von 80 % Ähnlichkeit als willkürlich. Ähnliches gilt für eine Studie von Modiba et al. [MPH16], in der die Ergebnisse von Werkzeugen zur Plagiatserkennung auf realen Daten mit einer manuellen Überprüfung auf Plagiate verglichen werden. Die Untersuchung legt den Fokus ebenfalls auf die Rate der Erkennung der Plagiate und liefert keine Erkenntnisse darüber, welche Ähnlichkeitswerte als (un)verdächtig angesehen werden können.

Pang und Vahid [PV23] gehen den umgekehrten Weg und nehmen die hohe Rate an vermeintlich nahezu identischen Lösungen zum Anlass, Programmieraufgaben gezielt so zu erweitern, dass die zu erwartende „natürliche“ Ähnlichkeit reduziert wird. Die Arbeit liefert damit zumindest einige Datenpunkte, welcher Grad an Ähnlichkeit in Abhängigkeit von der Komplexität der Aufgabe bzw. dem Umfang der Lösungen erwartet werden kann. In eine ähnliche Richtung geht eine Untersuchung von Haan und Striewe [HS21], die unter anderem den Einfluss der durchschnittlichen Länge einer Lösung auf die Ähnlichkeit untersucht, dabei jedoch zu uneindeutigen Ergebnissen kommt.

2 Datengrundlage und Methodik

Die verwendeten Daten stammen aus den Klausuren einer Programmierungsvorlesung in Java für Erstsemester. Hierbei wurden jeweils Haupt- und Nachtermin der Wintersemester 2020, 2021, 2022 untersucht. Im Jahr 2020 wurden bedingt durch die Corona-Pandemie die Klausuren zulassungsfrei, online und ohne Aufsicht als Freiversuch zu Hause geschrieben. In den Jahren 2021 und 2022 fanden die Klausuren in den Universitätsräumen unter Aufsicht statt und benötigten eine Zulassung, die durch vorherige Testate erworben wurde. Um Plagiaten bei der Onlineklausur im Jahr 2020 vorzubeugen, wurden vier unterschiedliche Versionen der Klausur erstellt. Alle Klausuren wurden mithilfe der Entwicklungsumgebung Eclipse durchgeführt. Die Studierenden erhielten Vorlagen in Form von Java-Quellcode-Dateien. Dieser Quellcode besaß Hilfsmethoden und zu

verwendende Objekte. Die Methodensignaturen der zu bearbeitenden Methoden waren vorgegeben. Die Lösungen wurden mittels eines E-Assessment-Systems eingereicht. Es gab für die Studierenden während der Klausur keinerlei Feedback zu ihren Einreichungen.

2.1 Auswahl der Aufgaben

Um einen sinnvollen Vergleich zu ermöglichen, wurden für dieses Paper thematisch ähnliche Aufgaben mit ausreichend großem Programmieranteil und Bearbeitungen ausgewählt. Zwar gab es in jeder Klausur eine Aufgabe zu Lambda-Ausdrücken, jedoch war diese mit fünf Zeilen Code zu lösen, welche für eine korrekte Lösung einen sehr eingeschränkten Lösungsraum aufwiesen. Bei einer Modellierungsaufgabe mussten lediglich Klassen mit Attributen und Getter-/Setter-Methoden mit vorgegebenen Bezeichnern erzeugt werden, was für die Plagiatsprüfung unbrauchbar war.

Alle Klausuren enthielten eine Aufgabe zu Arrays. Eine Aufgabe mit verketteten Listen war in allen Klausuren bis auf den Haupttermin 2022 vorhanden. In beiden Fällen war ein ausreichend großer Anteil von selbst verfasstem Quellcode gegeben, um einen Vergleich durchzuführen. Die Menge an studentischen Lösungen wurde um Leerabgaben bereinigt. Die sich insgesamt ergebenden Zahlen an verwertbaren Abgaben sind in Tab. 1 ersichtlich.

Tabelle 1: Anzahl der Abgaben je Aufgabe und Klausur. Aufgabenvarianten (nur im Jahr 2020 eingesetzt) sind zusammengefasst.

Abgaben	2020 HT	2020 NT	2021 HT	2021 NT	2022 HT	2022 NT
Array	321	175	165	93	173	69
Liste	196	125	172	93	---	69

2.2 Auswahl der Werkzeuge

Von den frei verfügbaren Softwarewerkzeuge zur Plagiatserkennung für Java-Quellcode wurde JPlag[4] aufgrund seiner großen Verbreitung und guten Erkennungsquote [WW12] ausgewählt. Zur Analyse des Quellcodes verwendet JPlag eine auf Token basierende Technik [PM02]. Plaggie[5] kann nur Java-Quellcode bis zur Version 1.5 verarbeiten und wurde daher ausgeschlossen. SIM[6] ließ sich auf den aktuellen Systemen nicht mit vertretbarem Aufwand installieren. MOSS[7] musste ausgeschlossen werden, da die Daten auf fremde Server hochzuladen waren. Sherlock[8] verwendet neben textbasierten auch eine auf Token basierende Technik zur Quellcodeanalyse [JL99] und wurde daher als weiteres Werkzeug verwendet. Beide Tools unterstützen die Verwendung von Vorlagen, so dass nur der Quellcode untersucht wird, welcher nicht in der Vorlage enthalten ist.

[4] https://github.com/jplag/JPlag
[5] https://www.cs.hut.fi/Software/Plaggie
[6] https://dickgrune.com/Programs/similarity_tester
[7] https://theory.stanford.edu/~aiken/moss/
[8] https://warwick.ac.uk/fac/sci/dcs/research/ias/software/sherlock/

2.3 Verwendete Methode

Zunächst wurden alle Einreichungen anonymisiert und für das Jahr 2020 nach Varianten getrennt, um den unterschiedlichen Bezeichnern der Varianten Rechnung zu tragen. Alle Codevorlagen wurden in den Tools hinterlegt. Jede Variante und Klausur wurde mit identischen Einstellungen der Tools verarbeitet. In JPlag wurde die vorausgewählte Methode der Durchschnittsberechnung und die Sensitivität von 9 verwendet. In Sherlock wurde die empfohlene Kombination aus „Original“, „No Comments and Normalised“ und „Tokenised“ verwendet. Das Resultat beider Tools war eine Menge an Paaren. Für jedes Paar gibt eine Kennzahl die Ähnlichkeit beider Einreichungen zueinander an. Sherlock gibt hierbei nur Paare mit einer Kennzahl über 0 aus, während JPlag alle Paare ausgibt.

3 Ergebnisse

Aufgrund der Technik des Paarvergleichs ergibt sich eine hohe Anzahl an Werten, die zudem aufgrund der unterschiedlichen Techniken nicht unmittelbar zwischen JPlag und Sherlock vergleichbar sind. Zur besseren Übersicht wurden die gemeldeten Paare anhand ihrer Werte in jeweils zehn Gruppen zusammengefasst. Damit die Ähnlichkeitsverteilung zwischen den Klausuren trotz der unterschiedlich hohen Zahl an Abgaben verglichen werden kann, wird jeweils das prozentuale Verhältnis zwischen der Anzahl der Paare je Gruppe und der Gesamtzahl der gefundenen Paare ermittelt und im Diagramm angegeben.

3.1 JPlag

In Tab. 2 und Abb. 1 ist die Ähnlichkeitsverteilung von JPlag für die Arrayaufgaben ersichtlich. Für die beiden Onlineklausuren im Jahr 2020 konnten nur die Einreichungen innerhalb einer Klausurvariante verglichen werden; dadurch ergibt sich eine geringere Paaranzahl. Einige Einreichungen konnten aufgrund von Syntaxfehler oder zu wenig Quellcode nicht von JPlag verarbeitet werden, daher gibt die zweite Spalte in Tab. 2 und Tab. 3 die Anzahl der von JPlag verarbeiteten Einreichungen an. Mit Ausnahme der Klausuren 2021 NT und 2022 HT weist der größte Teil der Abgaben kaum Ähnlichkeiten auf. Zwischen 20 und 50 % ergibt sich eine kleine Häufung. Im höchsten Bereich der Ähnlichkeit sind deutlich weniger Verdachtsfälle.

Die Ähnlichkeitsverteilung in der Klausur 2022 HT weicht von den anderen deutlich ab. Dies hängt mit den Aufgabenstellungen zusammen. Eine von drei Teilaufgaben bestand darin, drei mit unterschiedlich Werten gefüllte Arrays zu erstellen, um eine selbst erstellte Methode damit auf Korrektheit zu testen. Aufgrund der Token-Technik von JPlag werden unterschiedliche Zahlenwerte im Array als identisch erkannt. Auch die zweite Teilaufgabe erhöhte die Ähnlichkeitsquote, da sie die Implementierung eines ausführlich in Vorlesung und Übung behandelten Algorithmus erforderte.

Die Arrayaufgabe der Klausur 2021 NT weist als einzige einen zweistelligen relativen Ähnlichkeitswert von 13 % bzw. 560 Paaren im Bereich 90 bis 100 % auf. Die

Aufgabenstellung war, aus einem gegebenen eindimensionalen Array alle Strings, welche kürzer als ein übergegebener Parameter sind, in einem neuen Array zurückzugeben. Diese Aufgabe wurde zuvor nicht behandelt. Auch bei manueller Überprüfung ergibt sich eine sehr hohe Ähnlichkeit. Hier muss die Aufgabenstellung derart einengend sein, dass die Studierenden von selbst auf die gleiche Anweisungsabfolge gekommen sind, welche zugleich korrekt und minimal ist.

Bei den Aufgaben zum Thema verkettete Liste zeigt Tab. 3 deutlich geringere Ähnlichkeiten, so dass hier noch mehr Paarvergleiche eine geringere Ähnlichkeit als 10 % erreichen. Auch bei den Listenaufgaben ergibt sich eine zweite kleine Häufung zwischen 10 und 40 % welche bei 20 % den höchsten Wert erreicht.

Die Einreichungen zu Listen haben ähnlich viele Quellcodezeilen wie zu Arrays, somit kann dies nicht ausschlaggebend für die geringeren Ähnlichkeiten sein. Arrays lassen jedoch durch die spezifische Syntax möglicherweise weniger Lösungsvarianten zu.

Tabelle 2: JPlag Ähnlichkeitsverteilung mit Anzahl der Paare bei den Arrayaufgaben

Aufgabe Array	Einreich-ungen	Anzahl Paare	Ähnlichkeit in Prozentpunkten									
			0-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80	81-90	90-100
2020 HT	300	11367	9461	0	37	235	630	539	223	47	25	170
2020 NT	138	2362	1963	5	67	113	124	62	17	4	3	4
2021 HT	154	11781	5209	288	2173	1849	1015	791	268	118	57	13
2021 NT	93	4278	2627	0	6	325	232	196	63	125	144	560
2022 HT	165	13530	1316	1160	1877	2759	1683	1993	1390	719	400	233
2022 NT	67	2211	1156	126	293	165	197	94	81	59	26	14

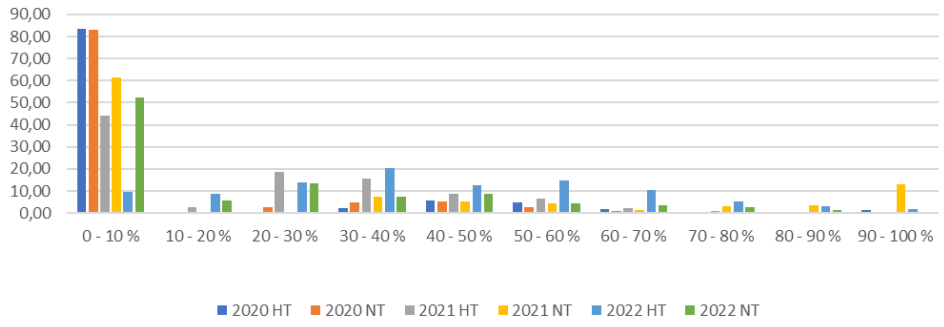


Abbildung 1: JPlag Ähnlichkeitsverteilung in Prozent bei den Arrayaufgaben

Tabelle 3: JPlag Ähnlichkeitsverteilung mit Anzahl der Paare bei den Listenaufgaben

Aufgabe Liste	Einreich-ungen	Anzahl Paare	Ähnlichkeit in Prozentpunkten									
			0-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80	81-90	90-100
2020 HT	191	4050	3951	0	3	10	17	15	13	14	11	16
2020 NT	78	762	678	1	25	41	8	5	3	0	1	0
2021 HT	166	13695	8172	1050	2552	1163	471	180	72	27	7	1
2021 NT	92	4186	1781	507	1069	431	271	95	23	9	0	0
2022 NT	67	2211	2071	40	68	23	2	2	3	2	0	0

3.2 Sherlock

Da Sherlock keine Paare mit 0 Ähnlichkeit ausgibt und Werte unter 10 äußerst selten sind, wurden alle Fälle mit einer Kennzahl kleiner 20 zusammengefasst. Die Kennzahl ergibt sich aus der Summe der einzelnen verwendeten Methoden. Codezeilen, welche in der Vorlage vorhanden sind, werden nicht als identisch gewertet. Die Kennzahl einer einzelnen Methode ist die Menge an identischen Zeilen im Vergleich zur Gesamtmenge der Zeilen. Diese Gesamtmenge wird nicht um die Zeilen der Vorlage reduziert, womit sich bei größeren Vorlagen geringere Maximalwerte der Methoden ergeben. Somit lassen sich verschiedene Aufgaben nur bedingt mit absoluten Kennzahlen vergleichen. Das Augenmerk sollte auf der Verteilung im gegebenen Spektrum der einzelnen Klausur liegen. Tab. 4 und Abb. 2 zeigen die Menge an Verdachtsfällen und die Ähnlichkeits-verteilung von Sherlock für die Arrayaufgaben.

Bis auf 2020 HT ist bei allen Klausuren ein steter Abfall der Menge an Fällen ab 30 zu erkennen. Die größere Menge an Fällen im 30er Bereich der Klausur 2020 HT kommt zustande, da am Ende der Aufgabe ein Array mittels einer Schleife befüllt und zurückgegeben werden musste, was keine variablen Lösungen zulässt. Somit fallen korrekte Lösungen in diesen Bereich. Die wesentlich größere Gesamtmenge an Fällen in der Klausur 2022 HT wurde bereits in Abschnitt 3.1 erläutert. In der Klausur 2021 HT war die Methodensignatur in der Aufgabenstellung vorgegeben, allerdings nicht in der Vorlage vorhanden, weshalb Sherlock eine große Menge an Fällen im Bereich bis 30 aufweist. Die vermehrten Fälle im Bereich 70+ dieser Klausur wurden bereits in Abschnitt 3.1 dargelegt. In 2021 NT befand sich eine wesentlich größere Vorlage, als in allen anderen Klausuren, weswegen hier insgesamt niedrigere Werte erzielt wurden.

Tabelle 4: Sherlock Anzahl der Verdachtsfälle bei den Arrayaufgaben

Aufgabe Array	Einreich-ungen	Anzahl Fälle	Ähnlichkeit in Punkten									
			< 20	20-29	30-39	40-49	50-59	60-69	70-79	80-89	90-99	100+
2020 HT	321	2522	40	659	1251	443	104	10	6	5	2	2
2020 NT	175	1759	237	658	402	235	128	63	24	8	2	2
2021 HT	165	6839	1477	2465	1356	711	405	217	136	45	22	5
2021 NT	93	1653	482	676	402	42	34	10	6	1	0	0
2022 HT	173	9380	4561	2040	1485	742	323	131	54	28	10	6
2022 NT	69	1350	292	444	280	132	105	50	26	13	6	2

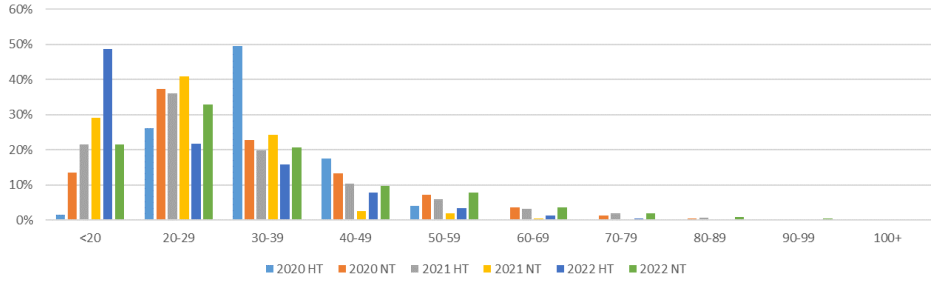


Abbildung 2: Sherlock Ähnlichkeitsverteilung in Prozent bei den Arrayaufgaben

Tabelle 5: Sherlock Anzahl der Verdachtsfälle bei den Listenaufgaben

Aufgabe Liste	Einreich-ungen	Anzahl Fälle	Ähnlichkeit in Punkten									
			< 20	20-29	30-39	40-49	50-59	60-69	70-79	80-89	90-99	100+
2020 HT	196	2	1	0	1	0	0	0	0	0	0	0
2020 NT	125	15	14	0	0	0	0	1	0	0	0	0
2021 HT	172	884	846	36	2	0	0	0	0	0	0	0
2021 NT	93	225	221	3	1	0	0	0	0	0	0	0
2022 NT	69	4	4	0	0	0	0	0	0	0	0	0

Bei den Aufgaben zum Thema verkettete Liste zeigt Tab. 5, dass Sherlocks Umgang mit Vorlagen bei Listen zu Schwierigkeiten in der Erkennung von Verdachtsfällen führt. In 2020 HT musste gegebener Code korrigiert werden, womit die Zeilen ignoriert wurden. 2020 NT besaß sehr viel vorgegebenen Code und nur eine kleine Methode, die zu implementieren war. In 2021 HT und NT gab es eine schlankere Vorlage als in den anderen Jahren und die Studierenden mussten unter anderem eine printList()-Methode implementieren, deren eingeschränkter Lösungsraum in der großen Menge an Verdachtsfällen unter 20 resultiert.

4 Diskussion

JPlag und Sherlock weisen bei Listen dieselben Ergebnisse auf. Die Menge an Verdachtsfällen ist bei den drei Klausuren 2020 HT & NT und 2022 NT äußerst gering, während bei den Klausuren von 2021 größere Mengen mit leichten Verdachtsfällen bestehen. Sowohl bei den Array- wie Listenaufgaben, liegt die große Mehrheit aller Paare insgesamt im Bereich 0–10 % Ähnlichkeit. In Anbetracht der kurzen Lösungen von ca. 10 bis 20 Zeilen mit vorgegebenen Methodensignaturen und daraus resultierendem geringen Varianzbereich der möglichen Programabläufe ist dies beachtlich und deutet darauf hin, dass eine überdurchschnittliche Häufung hoher Ähnlichkeiten in der Tat ein geeignetes Verdachtsmoment darstellt. Allerding wurde in der Arrayaufgabe der Klausur 2021 NT bei JPlag eine solche Häufung im 90–100 % Bereich gefunden. Da diese Klausur unter Aufsicht stattfand zeigt dies, dass auch ein solch hoher Wert nicht zwangsläufig auf Plagiate zurückzuführen ist. Vielmehr lernen die Studierenden im Semester einen bestimmten Stil oder eine Denkweise, um die behandelten Aufgaben zu lösen. Dies schlägt sich auch unter Aufsicht in sehr ähnlichen Einreichungen nieder.

Da die Klausuren aus 2020 in vier Varianten aufgeteilt waren, ergeben sich weniger Paare. Allerdings waren aufgrund der Zahl der Studierenden (selbst ohne die Erwartung von Betrugsversuchen aufgrund der fehlenden Aufsicht) in etwa doppelt so viele Verdachtsfälle wie gefunden wurden zu erwarten. Ein Grund für die Abweichung kann der zulassungsfreie Freiversuch sein, der dafür sorgte, dass sich viele Studierende wesentlich schlechter auf die Klausur vorbereitet hatten. Viele Einreichungen erfüllten die Aufgabenstellung nicht einmal ansatzweise und wiesen weder Ähnlichkeiten zu korrekten Lösungen noch zu anderen Fehlversuchen auf. Die Arrayaufgabe von 2020 NT ist zudem wesentlich freier und komplexer als andere gestellte Aufgaben, was zu mehr Individuallösungen führte. In solchen Fällen ist es möglicherweise zielführender, nach charakteristischen,

ungewöhnlichen Fehlern oder stilistischen Eigenheiten zu suchen, die in der Normalisierung durch die Werkzeuge untergehen.

5 Fazit und Ausblick

Die Ergebnisse zeigen, dass die Ähnlichkeitsverteilung maßgeblich von der Aufgabenstellung abhängt. Aus den variierenden Verteilungen können keine allgemeingültigen Schlüsse gezogen werden. Eine besondere Auffälligkeit im Vergleich der überwachten Präsenzklausuren zu den nicht überwachten Prüfungen konnte nicht festgestellt werden.

Sherlock und JPlag können als Tools zur Plagiatserkennung lediglich einen ersten Verdacht liefern, dem manuell nachgegangen werden muss. Beide Tools benötigen spezifische Rahmenbedingungen, um eindeutige Ergebnisse liefern zu können, welche in den untersuchten Daten nur zum Teil vorlagen. Entsprechend fällt die Definition schwer, ab wann eine Einreichung ähnlich zu einer anderen Einreichung ist. Ob beide Tools mit anderen Einstellungen und Rahmenbedingungen vergleichbarer werden, kann an dieser Stelle nicht beantwortet und muss in weiteren Arbeiten untersucht werden.

Literaturverzeichnis

[PV23] Pang, A.; Vahid, F: Variability-Inducing Requirements for Programs: Increasing Solution Variability for Similarity Checking. In: Proc. 28th annual ACM conference on Innovation and Technology in Computer Science Education (ITiCSE), 2023.

[HS21] Haan, T.; Striewe, M.: On the Influence of Task Size and Template Provision on Solution Similarity. In: Proc. 5. Workshop Automatische Bewertung von Programmieraufgaben (ABP), S. 51–58, 2021.

[St21] Strickroth, S.: Plagiarism Detection Approaches for Simple Introductory Programming Assignments. In: Proc. 5. Workshop Automatische Bewertung von Programmieraufgaben (ABP), S. 43–50, 2021.

[MPH16] Modiba, P.; Pieterse, V.; Haskins, B.: Evaluating plagiarism detection software for introductory programming assignments. In: Proc. CSERC ’16. ACM, 2016.

[NJK19] Novak, M.; Joy, M.; Kermek, D.: Source-code Similarity Detection and Detection Tools Used in Academia. TOCE 19/3, S. 1–37, 2019

[WW12] Weber-Wulff, D.: Collusion Detection System Test Report 2012. URL: https://plagiat.htw-berlin.de/collusion-test-2012/ (besucht am 07.06.2023)

[PM02] Prechelt, L.; Malpohl, G.: Finding Plagiarisms among a Set of Programs with JPlag. Journal of Universal Computer Science. 8, S. 1016–1038, 2002

[JL99] Joy, M.; Luck, M.: Plagiarism in programming assignments, in *IEEE Transactions on Education*, vol. 42, no. 2, pp. 129–133, May 1999

Identification of Hidden Structures in the Reference Network of E-Assessment Systems

Viola Weickenmeier[1] and Sven Strickroth[2]

Abstract: Using e-assessment systems, such as automated graders or automated feedback systems, is quite common in programming courses. Various tools have been developed to support students and teachers in learning and teaching programming. For overviews, comparisons of tools, and the identification of categories, a number of literature surveys and reviews have been carried out manually. This study does not try to find new systems, but uses (social) network analysis and citation data to identify important/influential systems as well as connections and similarities between systems in an existing corpus. The references were automatically extracted from the scientific publications related to these systems. Using these analyses, different types of communities and influential systems could be identified. Furthermore, there seem to be two types of references, those that simply mention a system and those that discuss specific features in more detail.

Keywords: reference analysis; e-assessment systems; (social) network analysis; modularity clustering

1 Motivation

Systems to automatically assess and provide feedback on student submissions are often used in programming courses. Over the past decades, many systems with different features and goals have been developed to support instructors and students by many researchers [SS22].

Literature reviews are a common practice for gaining insights into current approaches and topics. Typical goals of such reviews are to identify relevant literature and to analyze the described approaches and results often regarding very focused aspects. Sometimes there are meta-data analyses such as “How many papers were published per year?” or attempts to identify different categories of such systems (e. g., [SFB16; SS22]).

There is, however, no analysis of the reference network resulting from linking citations between scientific publications or associated systems for e-assessment systems. A network analysis can provide a holistic view, considering the relationships and contextual information surrounding citations [WVN10]. By identifying clusters of closely connected systems, central systems, influential authors, hidden patterns, and latent research communities can be uncovered. Such insights can foster dialogue, and promote the exchange of ideas.

In this paper preliminary results on a reference network analysis are presented to answer the following research questions: What are the most influential systems?, What (type of) clusters can be identified?, and Can regional communities be identified by their citations?

[1] LMU Munich, Institut für Informatik, Oettingenstr. 67, 80538 München, viola.weickenmeier@campus.lmu.de
[2] LMU Munich, Institut für Informatik, Oettingenstr. 67, 80538 München, sven.strickroth@ifi.lmu.de, https://orcid.org/0000-0002-9647-300X

doi:10.18420/abp2023-5

This paper is organized as follows: First, related work is reviewed and the research gap identified. Second, the methodology of this research is described and the results are presented. The paper concludes with a discussion of the results, a conclusion, and an outlook.

2 Related Work

There are several literature reviews on e-assessment systems, automatic graders, ITS, and hint systems that analyze student submissions, provide feedback, and/or grade them ([SS22] provides an overview): There are reviews such as Keuning et al. [KJH16] which categorizes 69 tools based on the type of feedback (cf. [Na13]). Saito et al. [Sa17] reviewed 43 tools and proposed a pedagogical taxonomy. De Souza et al. [SFB16] reviewed 30 systems and provided a classification on (semi-)automatic assessment type, student or teacher centered approaches and speciality (contest, quiz, software testing). There are also reviews focusing on specific aspects such as usage of AI in assessment systems [Le13]. All categories are developed manually and might not unveil hidden ones. Several approaches exist to discover topics or categories such as an analysis of the used keywords or topic modelling (e. g., [GS04]). These text mining methods, however, rely on the wordings used in the papers.

All these approaches ignore the reference network. Commonly used for linking data are author-based analyses such as co-authorship [GS05]. Here, two authors are linked if they published at least one paper together. Major limitations are, however, that collaborations are not represented sufficient enough through co-authorships [GS05] especially in this domain as most e-assessment systems seem to be developed independently (often as part of theses) [SS22], and are not suitable for finding cross-country connections between systems [Li05].

With the increasing use of (Social) Networking Analysis for investigating structures, there are different methods tested and validated for comparing scientific papers such as citation analysis [BK10]. This method focuses on references instead of content or authorship and is capable of finding the most important papers of a research topic [Wh04] as well as clustering or mapping of bibliographic data [WVN10].

Until now, there has been no work published that focuses on citation analysis to investigate structures and connections between e-assessment systems. Hence, there is no analysis of the reference network to get deeper insights, e. g. to find (new) classifications or to identify influential systems. It also remains unclear whether existing categorizations also reflect research communities or whether there are different research communities working on similar topics without knowing from each other. This research gap is addressed in this paper.

3 Method

The goal of this research is to analyze the reference-network of the papers and systems included in the corpus of [SS22]. To build the reference network, a tool was developed that

automatically analyzes PDF files and extracts the references. These data are matched with the meta-information in form of a BibTeX-file that contains all publications for all systems of in the corpus. The data used here, is based on an optimized version of the tool developed in [We23], and will be made available for download (also the data).[3] The nodes are the systems (no paper deals with multiple systems). The network is a directed graph, however, it was interpreted as an undirected graph if not stated otherwise. The graph analysis was conducted using Gephi.[4] References to the mentioned systems can be found in [SS22].

For the first part of the analysis, techniques from graph theory and (social) network analysis are used. Apart from general characteristics such as degrees, centrality is measured. Centrality indicators assign a ranking to nodes within a graph corresponding to their position in the graph [Ko05]. Two centrality measures are calculated: First, the Closeness Centrality (CC) is calculated, which is the inverse of the sum of the lengths of all shortest paths between a node and all other nodes in the graph. Thus, the more central a node is, the closer it is to all other nodes. Second, the Betweeness Centrality (BC) is calculated, which is the number of shortest paths from every node to every other node that a specific node is on. Hence, the BC indicates how often a node acts as a bridge along the shortest paths between different systems. For the second part of the analysis, the modularity method was used for clustering based on the works [Bl08; LDB08] as implemented in Gephi.

4 Results

The reference graph consists of 178 nodes and 340 edges (i. e., references; self-loops are excluded). The graph is not fully connected and consists of 32 components. There is one component comprising 146 systems, one component consisting of the two systems `HackerRank` and `Senecode`, and 30 unconnected systems. The diameter (longest distance between two connected nodes) is 8, the density is 0.02, and the average path length is 3.3. Median in and out degrees are both 1 and average in and out degrees are both 1.9. The most referenced systems (i. e., most incoming edges) in the graph are `WebCat` with 30 references, followed by `Coursema(rk|st)er` with 29 references (cf. Tab. 1). 62 systems have no outgoing edges and 81 no incoming edges.

Tab. 1 shows the top-10 systems ordered by their reference frequency (left) and betweeness centrality (middle). The top-3 systems are the same in both cases (namely `WebCat`, `CourseMarker` and `Singh`'s system), however, only three further systems are in both top-10 (namely `AutoLEP`, `Mooshak`, and `Fitchfork`). All closeness centralities are below .5 except for Senecode and Hackerrank (the cluster of two, hence CC=1). On the right side Tab. 1 shows the top-10 systems ordered by the directed betweeness centrality. Again, WebCat is the system with the highest value, however, the following two systems are of German origin (`JACK` and `ASB`). Overall, `Praktomat` is the German system with the most references (11) and `JACK` (referenced 6x) with the largest closeness (.38) and betweeness (703) centralities.

[3] `https://systemscorpus.strickroth.net`
[4] `https://gephi.org/`, version 0.10

System	deg	CC	BC	System	BC	System	dir. BC
WebCat (2003)	30	0.47	3468	WebCat	3468	WebCat	1291
CourseMarker (1998)	29	0.43	1821	CourseMar...	1821	JACK	1238
Singh-name (2013)	19	0.38	1648	Singh-name	1648	ASB	982
Mooshak (2001)	16	0.38	796	Fitchfork	799	CourseMar...	976
AutoLEP (2004)	12	0.38	607	Mooshak	796	eduComp...	947
Praktomat (1999)	11	0.32	342	JACK	703	Mooshak	711
JITS (2003)	10	0.33	498	Ask-Elle	641	PABS	468
Fitchfork (2006)	9	0.40	799	AutoLEP	607	eduJudge	245
Progtest (2011)	7	0.39	386	DsLab	595	Fitchfork	211
GAME (2004)	7	0.33	299	Galan-name	505	Progtest	207

Tab. 1: Overview of the top-10 systems (first usage year in parenthesis, cf. [SS22]) sorted by the in degree (left), betweeness centrality (middle), and directed betweeness centrality (right)

The modularity method for clustering formed 39 clusters (modularity: 0.53). The largest of the 32 graph components consists of 8 clusters. In total, 9 out of the 39 clusters have at least two nodes, the biggest cluster consists of 29 nodes. The average number of systems is 4.6, the median is 1. Without the clusters consisting of exactly one system the average number of systems is 9. In the following three of these clusters are examined in more detail.

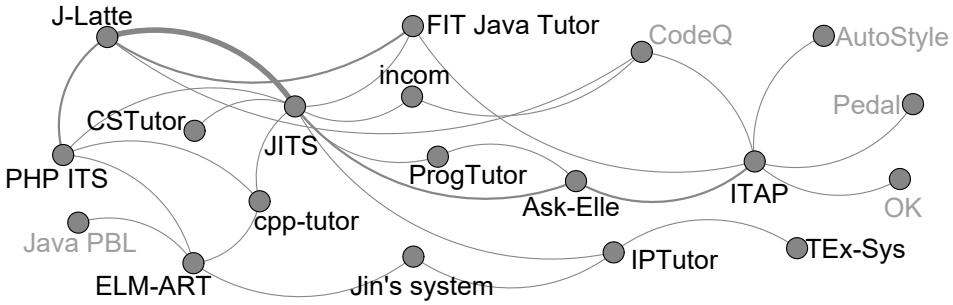


Fig. 1: Cluster 15 with 19 systems of which 14 are ITS (the nodes are systems)

In cluster 15 (cf. Fig. 1) out of the 19 systems 14 characterize themselves as Intelligent Tutoring Systems (ITS) and all but three use "intelligent tutor(ing system)" as a keyword in at least one paper. The most important system seems to be `JITS` here. An interesting case is `OK`, as this system self-characterizes as a hint system (the only one in the corpus) and not as an ITS. In the data set 21 systems self-characterize as ITS and 18 systems have at least one publication with the keyword "intelligent tutor(ing system)". The other systems are all distributed to different clusters. The filtered graph for self-characterized ITS is not fully connected. It contains, however, a subgraph of 14 connected systems (all black labelled nodes in Fig. 1) and 7 isolated systems (`COALA`, `iList`, `Burke`'s system, `Prutor`, `AWAT`, `M-PLAT`, and `WebIDE`). When the keyword "intelligent tutor(ing system)" is used as a additional filter,

J-Latte (no keywords), ITAP (“programming tutors”), ProgTutor (“tutoring”), and IPTutor (“programming tutor”) “vanish”, hence TEx-Sys is not connected any more to the other ITS.

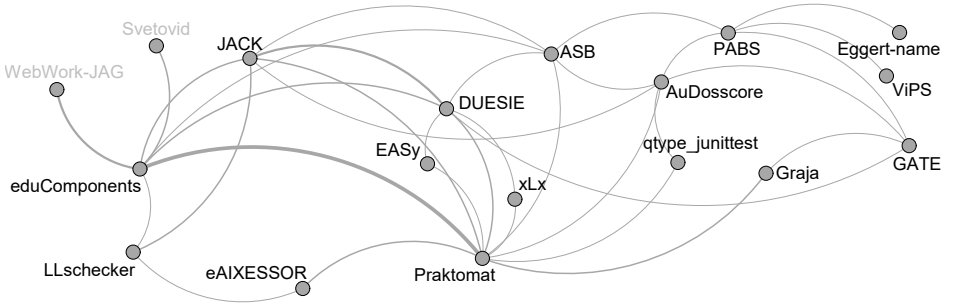


Fig. 2: Cluster 26 with 18 systems of which 16 have German origin (the nodes are systems)

Cluster 26 (Fig. 2) consists of 18 systems out of which 16 originate from German authors. There is also one system from Serbia (Svetovid) and WebWork-JAG from the USA included in this cluster. The most important system seems to be Praktomat. In the full data set there are 32 systems with a German origin. Half of the systems are included in this cluster. The German systems are included in 12 different clusters, five with more than one system (cluster 0: 1 system of 14 systems, 2: 2/29, 15: 3/18, 19: 1/17, 21: 2/24, 26: 16/18). There are three German ITS (ELM-ART, FIT Java Tutor, and incom) that are in the ITS cluster (cf. Fig. 1). Interestingly, the reference network of the 32 German systems is not connected if filtered. Unconnected (directly) to to other German systems are 14 systems including six systems Subato, OnExSy, IT4all, Burke’s system, ViPLab, and AutoTool that have no connection to any other system.

Cluster 28 consist of four systems namely MOE, Pythia, Code-Hunt, and Pex4Fun that form a “linked list” (in this order). Interestingly, the papers of Code-Hunt and Pex4Fun often share the very same main author and cover a similar approach. Code-Hunt and MOE share the keyword “(learning|educational) platform”.

Apart from the clusters, there are 15 systems in the corpus originating from Spain and 7 from Portugal which are the largest communities in Europe after Germany. The Spanish systems can be found in mainly three clusters (cluster 0: 4 systems of 14 systems, 21: 6/24, and 22: 3/21). Interestingly, the filtered graph contains a connected subgraph of 12 systems and three isolated systems (SAC, M-PLAT, and Munoz’s system). Four of the Portuguese systems are in cluster 21 (consisting of 24 systems) and the other three in three different clusters (2: 1/14, 6: 1/1, 22: 1/21). Only the systems code.org (isolated node, cluster 6) and CodeInsights are isolated, the others are all connected to the system Mooshak.

5 Discussion

A major limitation of this study is the restriction to papers and systems included in the corpus. The reason is the otherwise lack of annotated meta-data of publications. The way in which the corpus was compiled (cf. [SS22]) is probably also the reason for the quality of the clusters and the large number of isolated systems, as the vast majority of papers contain multiple references. However, by being strictly based on the corpus, the analysis can be reproduced easily. The extraction of the references was done automatically using an optimized version of a self-written tool. The accuracy was evaluated for the first prototype by randomly selecting 60 papers and checking their references manually (cf. [We23]). These papers contained 934 references in total of which 197 are to papers that are within the corpus of which 143 were correctly assigned. Out of the 737 other references, 2 were falsely assigned to papers within. The optimized version improved the recognition, but still no 100 % accuracy can be guaranteed as references itself contain mistakes or typos quite frequently (cf. [Ni07]). Finally, only the undirected graph was analyzed. The difference between the undirected and directed betweeness centrality indicates that the link direction also carries information (potentially e. g., evolution of ideas or systems that connect subcommunities).

Using the modularity method, three different types of reasonable clusters could be identified: First, a characteristic-based cluster that is heavily based on the system type (ITS). Here, closely related systems that do not identify themselves as ITS but use similar techniques are included. However, some ITS are not – maybe these are not well received by the ITS community or a "glue" system is missing in the corpus. Second, an origin-based cluster that is heavily based on the origin (Germany) of the authors. Still, there are half of the German systems in other clusters indicating that there are different communities or some researchers are not part of the community. Interestingly there is a fundamental difference between the German community and the Spanish and Portuguese ones. The German one forms its own cluster, having only 2 non-German systems in it. This is, however, not the case for the Spanish or Portuguese communities. While their systems are still well connected, they do not form their own clusters but instead are sub-groups in cluster 24 and 26. They refer to each other and also seem to be better embedded and referenced at the international level. This becomes evident in the most important Portuguese system `Mooshak` for which there are also papers reporting on usage from Spain, Finland, and India as well as a cooperation with authors from Greece. To some extend, the difference in the reference distribution is caused by the composition of the corpus which includes papers written in German but not papers written in Spanish or Portuguese. With more of such papers, the composition of the clusters may change, shifting the focus more on the language and nationality. Nevertheless, this difference shows that there seems to be a certain preference for German authors to reference German systems. Third, clusters that could not be explained based on their keywords, characteristics such as developed for MOOCs, or (self-)characterization. Especially keywords turned out not to be a good classifier as "assessment" or "automatic grading" can be found for nearly every cluster as top keywords. Additionally, keywords do not seem to be consistently used (cf. [SS22], e. g, the self-characterizes e-assessment `JACK` also used ITS in a publication).

One might think that the older a system is the more citations it gains. TEx-Sys and ELM-ART are the oldest systems in the corpus (both from 1992), but are not in the top-10 of the most cited systems. A reason could be that these two are ITS and not generic e-assessment systems. Most references are for systems such as CourseMarker (2002) and WebCat (2003). These two systems, seem to have founded a new trend and, therefore, might be referenced more often. Still, Praktomat (1999) seems to be the oldest German system and has indeed the most references followed by JACK (2008) that seems to have the most associated publications.

The references to the top 10 systems can be divided into two different groups, which we call quantitative and qualitative. CourseMarker was one of the first e-assessment systems. Most of the references to CourseMarker target two out of its four corresponding papers. The first is an introduction to the system, the other an usage report which also highlights the advantages of the system. Other systems that reference the first instead of the second paper are likely not interested in the properties of CourseMarker, instead referencing it just as an example for e-assessment systems in general. Additionally, CourseMarker has more links to other clusters as links within. That also means that many systems reference it without having much similarities, as they then would be in the same cluster instead. References to Mooshak tend to be more focused on the system and its properties. It is one of the first programming judges and is often cited for that reason. But instead of only mentioning it as one of the already existing systems like CourseMarker, it is mostly referenced with a longer and more detailed description. Also, most of its references are from within its own cluster.

6 Conclusion & Outlook

In this paper the reference network of the corpus established by [SS22] was analyzed. This preliminary study provides a first insight into the results. Based on a modularity clustering, three types of clusters could be identified: one that mainly contains ITS (characteristic-based cluster), one that mainly contains systems of German origin (origin-based cluster), and clusters that are not easily explainable based on common properties. Hence, no new categorizations could be unveiled. Additionally, for the most referenced systems an analysis of how the references look like in the paper was conducted. Two types of references could be identified: references just mentioning existing systems (quantitative) and references that intensively discuss system properties (qualitative). A further conclusion might be that “the” German community should write more in English to be reference-able from non-native speakers and network internationally.

Further research should try to use different clustering techniques such as clique percolation or fuzzy clustering where a node can be in multiple clusters and compare those with the clusters found here. This might address issues arising through the fact that systems might change their character over time. Also other characteristics such as the type of feedback or type of reference could be analyzed. Additionally, the analysis here is strictly bound to the system and papers included on the said corpus. It would be interesting to see whether external data on references could be used to extend the data set and search for clusters there.

Bibliography

[BK10] Boyack, K. W.; Klavans, R.: Co-citation analysis, bibliographic coupling, and direct citation: Which citation approach represents the research front most accurately? JASIST 61/12, pp. 2389–2404, 2010.

[Bl08] Blondel, V. D.; Guillaume, J.-L.; Lambiotte, R.; Lefebvre, E.: Fast unfolding of communities in large networks. JSTAT/10, P10008, 2008.

[GS04] Griffiths, T. L.; Steyvers, M.: Finding scientific topics. Proc. National Academy of Sciences 101/suppl_1, pp. 5228–5235, 2004.

[GS05] Glänzel, W.; Schubert, A.: Analysing scientific networks through co-authorship. In: Handbook of quantitative science and technology research. Springer, pp. 257–276, 2005.

[KJH16] Keuning, H.; Jeuring, J.; Heeren, B.: Towards a systematic review of automated feedback generation for programming exercises. In: ITiCSE. Pp. 41–46, 2016.

[Ko05] Koschützki, D. e. a.: Centrality Indices. In: Network Analysis. Springer, pp. 16–61, 2005.

[LDB08] Lambiotte, R.; Delvenne, J.-C.; Barahona, M.: Laplacian dynamics and multiscale modular structure in networks. arXiv preprint arXiv:0812.1770/, 2008.

[Le13] Le, N.-T.; Strickroth, S.; Gross, S.; Pinkwart, N.: A Review of AI-Supported Tutoring Approaches for Learning Programming. In: Proc. ICCSAMA. Springer, pp. 267–279, 2013.

[Li05] Liu, X.; Bollen, J.; Nelson, M. L.; Van de Sompel, H.: Co-authorship networks in the digital library research community. Information processing & management 41/6, pp. 1462–1480, 2005.

[Na13] Narciss, S.: Designing and evaluating tutoring feedback strategies for digital learning. Digital Education Review/23, pp. 7–26, 2013.

[Ni07] Nicolaisen, J.: Citation analysis. ARIST 41/1, pp. 609–641, 2007.

[Sa17] Saito, D. e. a.: Program learning for beginners: survey and taxonomy of programming learning tools. In: Proc. ICEED. IEEE, pp. 137–142, 2017.

[SFB16] de Souza, D. M.; Felizardo, K. R.; Barbosa, E. F.: A Systematic Literature Review of Assessment Tools for Programming Assignments. In: Proc. CSEET. Pp. 147–156, 2016.

[SS22] Strickroth, S.; Striewe, M.: Building a Corpus of Task-based Grading and Feedback Systems for Learning and Teaching Programming. iJEP 12/5, pp. 26–41, 2022.

[We23] Weickenmeier, V.: Identifying hidden structures among Research Papes for E-Assessment Systems, MA thesis, LMU Munich, Germany, 2023.

[Wh04] White, H. D.: Citation analysis and discourse analysis revisited. Applied linguistics 25/1, pp. 89–116, 2004.

[WVN10] Waltman, L.; Van Eck, N. J.; Noyons, E. C.: A unified approach to mapping and clustering of bibliometric networks. Journal of informetrics 4/4, pp. 629–635, 2010.

A Grammar and Parameterization-Based Generator for Python Programming Exercises

Philipp Peess[1], Annabell Brocker[2], Rene Roepke[2] and Ulrik Schroeder[2]

Abstract: As the importance of programming education grows, the demand for a sufficient number of practical exercises in courses also increases. To accommodate this need without significantly increasing the instructors' workload, a programming exercise generator capable of generating exercises for independent practice is considered. This research mainly focuses on determining suitable generation methods and creating a modular and extensible generator structure. The current generator implementation uses parameterization and a grammar-based generation approach in order to provide generated exercises directly to students in their programming environment. Furthermore, the generator can act as a foundation for further research and be extended with additional generation methods, creating the possibility of exploring artificial intelligence for the generation of programming exercises.

Keywords: Automatic Generation; Programming Exercises; Python; JupyterLab

1 Introduction

An important aspect of programming education is the provision of practical exercises, which can improve both the students' theoretical and practical knowledge [BE15], tailored to varying skill levels, possibly through personalization [Of17]. Programming courses often require a substantial number of practice tasks, but manual creation demands considerable time and expertise and may not meet individual learning needs, such as the need for numerous tasks related to a specific programming concept. The work aims to design and develop `pygenaix`, an automatic generator for programming exercises. Importantly, students must be able to self-evaluate their solutions to these exercises to enable independent practice.

For the design and development of the system, an introductory programming course at a German university, in which Python is introduced to a wide variety of non-CS students, was choses as a reference. In this paper, the following two research questions are addressed as part of the design and development of the exercise generator: **(RQ1)** “Which task generation methods are suitable for exercises used in an introductory programming course?” and **(RQ2)** “How can the generator implementation be extensible and easy to use in the course's learning environment?”. For future developments and further research, the resulting source code of the exercise generator was made openly available.[3]

[1] RWTH Aachen University, Templergraben 55, 52062 Aachen, Germany, philipp.peess@rwth-aachen.de
[2] RWTH Aachen University, Learning Technologies Research Group, Ahornstr. 55, 52074 Aachen, Germany, {a.brocker, roepke, schroeder}@cs.rwth-aachen.de, https://orcid.org/{0009-0007-6708-0892, 0000-0003-0250-8521, 0000-0002-5178-8497}
[3] pygenaix, `https://doi.org/10.17605/OSF.IO/TKHR3`, last accessed 15.09.2023

doi:10.18420/abp2023-6

2 Related Work and Approaches to Automatic Generation

When reviewing related work on the automatic generation of programming exercises, it was found that so far only little research was conducted in this field, and among the existing works, a range of different generation methods is addressed. As such, [ROS10] employed *parameterization* by developing a generator which replaces placeholders in both the task description and an example solution to create new tasks with small but intentional variations. In other works, *models* were transformed to programming exercises, ranging from mathematical formulas to decision trees and complex graph structures (e. g. [So21; TS18]). Similarly, [Ad19] used a *context-free grammar (CFG)* to generate a large number of example programs by defining a grammar covering the core structure of select Python programming concepts. Importantly, this grammar does not yet allow for the generation of a task description, but was nonetheless capable of generating suitable code fragments focused on specific topics. In [Sa22], an *artificial intelligence (AI)*, *OpenAI Codex*[4], was primed using example exercises consisting of a task description, a solution, tests and keywords, which describe the scenario and used programming concepts. When also providing new keywords, a new task could be generated. Lastly, [WM15] and [Ha18] used *fault injection* to modify a sample solution to a task description by introducing errors and then providing this faulty solution to students, which would have to find all errors. Consequently, this method is not suited for generating free programming exercises for the considered course.

Besides these works, the use of content generation methods in other domains was investigated, e. g. for the generation of block-based programming exercises [Ba17] and quizzes [Ku20] in an educational context, or the generation of unit tests [Se19] as a more general application domain. As these generation approaches usually only focus on one specific aspect, they are unsuitable for the generation of whole programming exercises.

Among the presented methods found in related work, parameterization-, model-, grammar- and AI-based approaches were the most promising candidates for further exploration. To determine the suitability of the generation methods in the context of the considered course, a requirement analysis was conducted, focusing on the needs of tutors (i. e. instructors) and students. As such, interviews with tutors were conducted to collect insights into the current task creation process as well as their perception of students' knowledge levels and common problems when dealing with programming exercises. Additionally, the course materials and exercises were analysed to identify suitable task types for automatic generation.

Firstly, parameterization provides a comparably low-complexity solution for generating programming tasks. It can easily be implemented on existing tasks by introducing placeholders and sets of values, thus creating task templates. Additionally, the process of replacing placeholder values is commonly used, even in other generation methods, and should therefore be implemented in any case as a possible post-processing step for the outputs of other generation methods. However, it also has major disadvantages, as the variability of the generatable tasks is strongly restricted by the fact that only specific sections of the input,

[4] OpenAI Codex, https://openai.com/blog/openai-codex, last accessed 30.05.2023.

i. e. the placeholders, can be changed. As such, another generation method offering more variability should be considered.

The remaining approaches offer a higher degree of variability, but also a high level of complexity. For model-based generation approaches, this may be controllable based on the specific model that is chosen for the generation process. More flexibility can be achieved using a grammar-based approach, as a grammar could also be used to generate different kinds of models. Additionally, a study showed that 93.1 % of students strongly agreed that the programs generated using a grammar-based approach can help them in practice and improve programming skills [Ad19]. Consequently, a grammar-based generation method was chosen as the second generation method the programming exercise generator offers.

AI-based generation methods were also considered, but due to the problems that occurred in the study presented in [Sa22], the reliability of the approach was questioned. In [Sa22], a sample set of 240 exercises generated by an AI was analysed and it was found that an example solution was missing in about 15 % of the cases and tests were missing in almost 30 % of the cases. Additionally, the sample solution passed the tests in less than a third of the cases it was generated with a solution, indicating that one of the two was faulty. Furthermore, it should be noted that AIs are generally black boxes, which makes it impossible to predict the next output for any given input. This makes it difficult to use AI as a direct source for programming exercises and makes an intermediate (human) control instance almost mandatory, which would likely introduce additional manual work. Also, when using a third-party solution, the training data is mostly unknown, making it hard to predict what the AI is capable of generating and may subsequently require manual rework to verify the usefulness of the task. Similar findings have been shown in another study where a purely AI-based approach with a large language model and prompts was chosen [SMB23]. While training an AI for programming exercise generation would generally be possible, it would be a complex problem and require sufficient training data. As such, we decided against implementing an AI-based generation approach in the initial version of the generator. However, with the recent advancements in large language models in AI that occurred during the research process of this work, this should be reevaluated. Additional research in this area should be conducted to investigate whether new AI-based tools are more capable of generating suitable and correct programming exercises.

3 Generator Design and Structure

The generator primarily focuses on creating exercises for novice programmers, emphasizing algorithm-oriented input-output tasks rather than complex software architecture challenges. Lecturers should be able to control the task topics themselves by specifying templates, so that, for example, tasks only focus on the topics of variable declaration and initialisation. A central goal in the design and implementation of a generator for programming exercises was to establish an extensible and modular software structure. Consequently, the generator was implemented in a plugin-based architecture, visualised in Figure 1.

As a first step, all external data was separated from the implementation of the generator itself. This includes the definition of the generation capabilities, such as data for replacing placeholders and the grammar definitions, as well as task blueprints, which act as templates for the generation process. These blueprints define the generation steps and additional metadata, such as topics a task may cover or potential filtering criteria like the task's difficulty. The resulting separation of the generator implementation and external data enables adding new content to the generator without modifying its implementation directly.

The generator itself is split into the main generator and multiple task generators. The main generator acts solely as a management component focused on loading and providing external data as well as controlling the generation process. When tasked with the generation of a new exercise, the main generator selects a blueprint and delegates the generation to the respective task generators. Each task generator implements a generation method, currently for both a parameterization- and a grammar-based generation approach. During the generation, multiple task generators can be used in sequence by using the output of one task generator as input for another task generator. This allows for the generation of more complex tasks while reducing the functionality each task generator has to provide, facilitating the introduction of new generation methods.

Parameterization: Fundamentally, the parameterization-based task generator replaces placeholder values with randomly selected entries from predefined data sets. These placeholders contain identifiers, making it possible to link related placeholders. Additionally, the data set entries are not single words or values, but instead take on the form of dictionaries to provide data as key-value pairs. This way, different grammatical forms and metadata providing further information on the entry, like associated topics, can be grouped. Conditions may then be used to select only specific entries matching certain criteria, allowing the creation of consistent and complex tasks. To facilitate the process of defining templates, additional predefined commands were added to the generator, supporting operations for grammatical adjustments, e. g. choosing *a* and *an*, or common operations like randomly generating

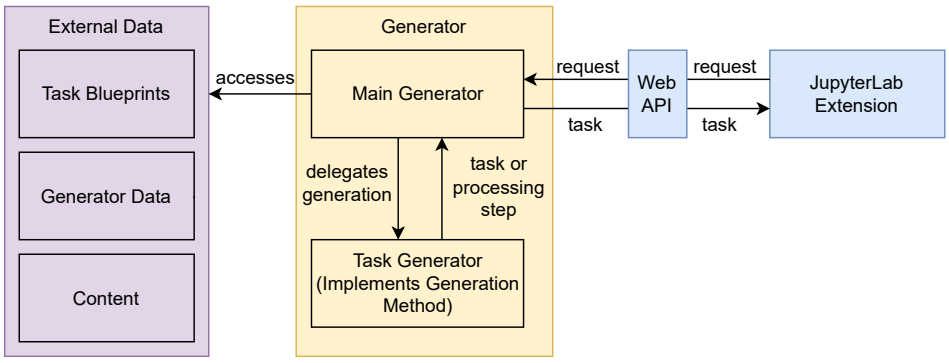


Fig. 1: System architecture visualising the different components and their interactions.

numbers. Furthermore, the parameterization-based generator supports the definition of any verification method, allowing for the automatic generation of tests and example solutions the students can then use to verify their solution.

Grammar-Based Generation: The foundation of the grammar-based generator is a modified version of a CFG (similar to [Si12]), which provides additional symbol types: *Meta symbols* may be defined to provide information for the post-processing steps and are not translated to an output value when evaluating the resulting symbols after the grammar evaluation; by using *direct symbols*, it is possible to define new content directly in the textual input instead of creating new grammar symbols, allowing for the creation of content for single tasks without bloating the grammar. During processing, both symbol types behave the same way a terminal symbol would and after resolving all non-terminal symbols, only terminals, meta symbols and direct symbols remain. Next, post-processing steps introduce variables and ensure newly generated functions and classes can be instantiated and called. This is done by analysing the current symbol sequence and searching for specific patterns that indicate the definition of functions and classes and determining their parameters and attributes. Any references to newly defined functions, e. g. function calls or class instantiations, may then be replaced with one of the generated elements. The processed grammatical representation of a Python program is then directly translated to source code. Importantly, the grammar also defines text values from which a task description can be derived, ensuring that the example solution matches the task description. Contrary to parameterization, the grammar-based approach allows only for the generation of an example solution and a matching task description. As such, an automatic verification is currently not possible. However, since practice tasks generally only encompass short tasks and code fragments, tools for generating unit tests from code, e. g. Pynguin[5], could potentially be used to generate tests automatically.

4 Available Interfaces for Stakeholders

To make the generator available for requests (e. g. from a learning environment), a web API was implemented using a *Flask*[6] server. It provides routes for all central functionality of the generator. Importantly, this includes a route for requesting new tasks from the generator which allows for configurations regarding the topic, the output format, requesting a download of the file as well as filtering criteria for the blueprint selection process. Additionally, it is possible to request information regarding the existing blueprints.

To simplify the process of generating exercises for students, a graphical user interface (GUI) in the form of a JupyterLab extension[7] was implemented. It groups different blueprints under topics and groups of topics under overarching categories corresponding to the course structure to guide students through choosing adequate tasks for their current knowledge

[5] Pynguin, https://pynguin.readthedocs.io/en/latest/index.html, last accessed 14.06.2023.
[6] Flask, https://flask.palletsprojects.com/en/2.3.x/, last accessed 31.05.2023.
[7] JupyterLab Extensions, https://jupyterlab.readthedocs.io/en/stable/extension/extension_dev.html, last accessed 05.06.2023.

level. For more control, attributes like the preferred difficulty of the task can be specified. Upon submission, a request is sent to the web API, which returns an ad-hoc-generated task matching the specification and opens it as a Jupyter Notebook in the lab environment.

Lastly, to facilitate the development and the addition of new blueprints, a testing interface for instructors (particularly tutors) was needed. In this context, a command line interface was implemented, simplifying access to the generator functionality with additional options.

This way, generation can be done in bulk and can be restricted to specific blueprints. As such, the interface facilitates testing new blueprints locally instead of having to move them to the server first. It also supports the management of additional blueprints for exam task generation which are usually not shared with students for preparation.

5 Evaluation and Future Work

To determine the generator's suitability, a preliminary user evaluation was conducted with tutors that already participated in the initial interviews. In a second round of interviews, the tutors provided feedback regarding both the generatable content as well as the process of defining new task types as blueprints. Regarding the former, the evaluation focused on five criteria: (1) the possibilities for personalization, (2) the support of storytelling, (3) the variability of the generated content and their fit regarding (4) foundational topics (e. g., loops, recursion) and (5) advanced problems (e. g., sorting algorithms). The tasks were generally received favourably regarding these criteria, but in some cases, especially (4), the tutors had vastly different opinions, which underlines the need for a comprehensive evaluation with students and particularly novice programmers.

The evaluation also uncovered some problems of the implementation that still have to be addressed. On the one hand, task descriptions should be improved in two central ways: Firstly, the storytelling provided in the tasks was problematic, as the generated code did not necessarily make sense in the described scenario and therefore could be perceived as confusing or misleading by the students. Secondly, task descriptions were, at least in the case of the grammar-based generation method, largely very direct descriptions of the solution. However, with a growing knowledge level as well as task difficulty, more abstraction of task descriptions regarding the sample solutions would be favourable. An additional limitation of the grammar-based approach is the imperative nature of the task descriptions derived directly from the imperative code, which does not match natural language well in many cases. On the other hand, the generation can currently lead to unfavourable results, e. g. unnecessary conditions like the comparison of a variable with itself. To counteract this, static code analysis should be introduced to automatically detect and prevent these cases.

Furthermore, the interviewed tutors generally agreed that the task definition process is understandable, but noted the need for documentation to simplify the blueprint creation process. Especially the grammar-based approach requires sufficient knowledge of the existing

symbols in order to provide valuable blueprints. Consequently, the improved documentation now contains an extensive overview of all allowed symbols. However, there could be more assistance for defining tasks, e. g. in the form of a GUI.

Finally, the generator provides a foundation for follow-up research and the introduction of further generation methods due to its modular and plugin-based architecture. The capabilities of different generation methods should be explored further, especially with regard to the potential benefits of AI. Another benefit of AI would be for post-processing generated tasks and task descriptions, e. g. to improve current problems with storytelling. Further, it would be interesting to investigate the benefits of AI for the generation of blueprints to facilitate the process of defining new task types. Also, AI could be used to generate test cases for generated tasks. Lastly, another application of AI would be the translation of generated task descriptions into different languages, as the generator currently only supports English tasks.

Regarding the general capabilities of the generator, the number of generated instances depends on the definition of the task blueprints, the parameterization data and the grammar definition. The number tasks directly correlates to the number of key-value pairs available and the number of values per key. The grammar may contain rules with infinite recursions, although this was limited to prevent overly long tasks. Additionally, the generator already provides a sample solution to the task, allowing learners to independently compare their solution with the sample solution. Automated feedback could be included by extending the static test or unit test interface. Existing packages, such as Pynguin[5], pycheckmate[8] or PyTA[9], could be used to generate static as well as unit tests.

6 Conclusion

This work presents the design and implementation of `pygenaix`, a programming exercise generator supporting parameterization and a grammar-based approach, and provides answers to the formulated research questions. To determine suitable generation methods **(RQ1)**, related works were analysed and evaluated, leading to the selection of parameterization and a grammar-based generation approach. With regards to an extensible and easy to use implementation **(RQ2)**, a structure was developed which allows for the addition of new generation methods as well as new generatable content. Furthermore, a web-based API for requests was developed, which is used to connect the generator to a JupyterLab extension providing a GUI for the generator, enabling the students to access the generator easily. Future work entails improvements to the implemented generation methods before moving on to exploring further methods, like AI for refining storytelling, generating programming exercises or deriving programming exercise blueprints from available exercises.

[8] pycheckmate, `https://doi.org/10.17605/OSF.IO/BR68W`, last accessed 28.08.2023
[9] PyTA, `https://github.com/pyta-uoft/pyta`, Last accessed: 28.08.2023

Bibliography

[Ad19] Ade-Ibijola, A.: Syntactic Generation of Practice Novice Programs in Python. In (Kabanda, S.; Suleman, H.; Gruner, S., eds.): ICT Education. Springer, Cham, pp. 158–172, 2019, ISBN: 978-3-030-05813-5.

[Ba17] Bart, A. C. et al.: BlockPy: An Open Access Data-Science Environment for Introductory Programmers. Computer 50/5, pp. 18–26, 2017, ISSN: 1558-0814.

[BE15] Berglund, A.; Eckerdal, A.: Learning Practice and Theory in Programming Education: Students' Lived Experience. In: Int. Conf. on Learning and Teaching in Computing and Engineering. LaTiCE'15, IEEE, New York, pp. 180–186, 2015.

[Ha18] Habibi, B. et al.: Using Fault Injection for Programming Task Generation. In (Auer, M. E.; Guralnick, D.; Simonics, I., eds.): Teaching and Learning in a Digital World. ICL'17, Springer, Cham, pp. 559–566, 2018, ISBN: 978-3-319-73204-6.

[Ku20] Kurdi, G. et al.: A Systematic Review of Automatic Question Generation for Educational Purposes. en, Artificial Intelligence in Education 30/1, pp. 121–204, 2020, ISSN: 1560-4306, URL: https://doi.org/10.1007/s40593-019-00186-y, visited on: 04/05/2023.

[Of17] Offutt, J. et al.: A Novel Self-Paced Model for Teaching Programming. In: 4th ACM Conf. on Learning @ Scale. L@S '17, ACM, New York, pp. 177–180, 2017, ISBN: 978-1-4503-4450-0, URL: https://doi.org/10.1145/3051457.3053978, visited on: 10/03/2022.

[ROS10] Radošević, D.; Orehovački, T.; Stapić, Z.: Automatic On-Line Generation of Student's Exercises in Teaching Programming. In: Central European Conf. on Information and Intelligent Systems. CECIIS'10, Varaždin, 2010, URL: https://papers.ssrn.com/abstract=2505722, visited on: 09/27/2022.

[Sa22] Sarsa, S. et al.: Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In: 18th ACM Conf. on Int. Computing Education Research. ICER '22, ACM, New York, pp. 27–43, 2022, ISBN: 978-1-4503-9194-8, URL: https://doi.org/10.1145/3501385.3543957, visited on: 09/27/2022.

[Se19] Serra, D. et al.: On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later. In: IEEE/ACM 16th Int. Conf. on Mining Software Repositories. MSR'19, IEEE, New York, pp. 121–125, 2019.

[Si12] Sipser, M.: Introduction to the Theory of Computation. Cengage Learning, 2012, ISBN: 978-1-133-18779-0.

[SMB23] Speth, S.; Meißner, N.; Becker, S.: Investigating the Use of AI-Generated Exercises for Beginner and Intermediate Programming Courses: A ChatGPT Case Study. 2023 IEEE 35th International Conference on Software Engineering Education and Training (CSEE&T)/, pp. 142–146, 2023, URL: https://api.semanticscholar.org/CorpusID:261433946.

[So21] Sovietov, P.: Automatic Generation of Programming Exercises. In: 1st Int. Conf. on Technology Enhanced Learning in Higher Education. TELE'21, pp. 111–114, 2021.

[TS18] Tiam-Lee, T. J.; Sumi, K.: Procedural Generation of Programming Exercises with Guides Based on the Student's Emotion. In: IEEE Int. Conf. on Systems, Man, and Cybernetics. SMC'18, IEEE, New York, pp. 1465–1470, 2018.

[WM15] Wakatani, A.; Maeda, T.: Automatic generation of programming exercises for learning programming language. In: IEEE/ACIS 14th Int. Conf. on Computer and Information Science. ICIS'15, IEEE, New York, pp. 461–465, 2015.

Parametrisierung von Haskell-Programmieraufgaben

Leon Koth und Janis Voigtländer[1]

Abstract: Wir berichten über Design und Verwendung einer Template- und Generator-DSL zur Parametrisierung von Übungs- und Klausuraufgaben einer Lehrveranstaltung zur Haskell-Programmierung.

Keywords: E-Learning, Haskell, Variabilität

1 Einführung

In [SVW19, Vortragsfolien unter `https://udue.de/HaskellABP19`] wurde ein in das E-Learning-System „Autotool“ [Wa17] integrierter Aufgabentyp zur Haskell-Programmierung vorgestellt. Es wurden vielfältige Konfigurationsmöglichkeiten beschrieben, etwa hinsichtlich der Feedback-Generierung mittels Compiler, Linter und dem deklarativen Testframework QuickCheck [CH00]. Eine konkrete Aufgabeninstanz wird definiert durch eine Textdatei mit diversen (teils YAML-basierten) Abschnitten zur Ansteuerung der verwendeten Tools, zur Bereitstellung einer Haskell-Programmdatei mit der eigentlichen Aufgabenstellung und Platz zum Lösen, und zur Angabe einer (gegebenenfalls versteckten) Testsuite. Zusätzlich wird eine Musterlösung angelegt, die alle konfigurierten Überprüfungen und Tests besteht.

Aus diversen Gründen kann es attraktiv sein, Aufgaben zunächst zu parametrisieren, um dann verschiedene Ausprägungen irgendwie „ähnlicher“ Aufgabeninstanzen erhalten zu können. Damit ist hier nicht gemeint, dass Tests mit zufälligen Werten durchgeführt werden, sondern eine tatsächliche Variabilität in der Aufgabenstellung selbst, in anderen Teilen der Konfiguration, und unter Umständen notwendigerweise auch in der passenden Musterlösung.

In einer Bachelorarbeit [Ko22] wurden entsprechende Funktionalitäten entworfen und implementiert, die seitdem sowohl im Übungsbetrieb als auch in einer Online-Klausur verwendet wurden. Der vorliegende Beitrag wird diese Funktionalitäten beispielgestützt besprechen und einige Aspekte diskutieren.

2 Verwandte Arbeiten

Automatische Systeme zum Stellen und Auswerten von Programmieraufgaben sind in der Informatiklehre weit verbreitet, die Verwendung von Aufgabengeneratoren in diesem Kontext offenbar weniger. So liefert ein aktueller Korpus praktisch eingesetzter Systeme [SS22] keinerlei Treffer zu Suchanfragen wie „Parametrisierung“, „Variabilität“ oder ähnlichen (auch

[1] Universität Duisburg-Essen, janis.voigtlaender@uni-due.de

doi:10.18420/abp2023-7

englischsprachigen) Begriffen, und liefert Treffer zu „Generierung“ lediglich hinsichtlich der Generierung von Testfällen, von Hinweisen und von Feedback, jedoch nicht von Aufgaben selbst. Gleichwohl gibt es Forschungsarbeiten zur Aufgabengenerierung im Bereich der Programmierlehre. Dabei kann es etwa um spezifisch und relativ ad-hoc entwickelte Generatoren für abgegrenzte Teilfelder des Programmierens gehen (zum Beispiel [We20] und [So21]) oder um allgemeiner einsetzbare aber mit weniger domänenspezifischer Semantik versehene Ansätze per Variantenbildung von ansonsten frei geformten Aufgaben.

Anforderungen und Besonderheiten des Weges über Variantenbildung wurden in [OG17] beispielgetrieben diskutiert. Eine frühe Umsetzung dieses Weges mit einfacher Template-Gestaltung aber noch recht eingeschränkten Variierungsmöglichkeiten ist [ROS10], hingegen flexibler hinsichtlich der Variationspunkte aber mit mehr Aufwand zur Template-Entwicklung stellt sich [CM19] dar. Besonders auf Wiederverwendbarkeit zielt [Ga19] ab, basierend auf einem systemübergreifenden Aufgabenformat [Re19, ProFormA 2.0] und algebraisch elaborierten Wertebereich-Produkten für Variationspunkte [Ga18].

Im vorigen Absatz genannte Arbeiten werden wir noch genauer zu Vergleichen mit unserem eigenen verwandten Vorgehen heranziehen. An dieser Stelle erwähnt, ansonsten aber außerhalb der Betrachtung dieses Artikels stehend, seien noch neue Möglichkeiten der Erzeugung und Variierung von Programmieraufgaben mittels generativer Sprachmodelle [Sa22].

Bei der Beschreibung in den kommenden Abschnitten verwenden wir zur Vereinheitlichung Begrifflichkeiten aus [Ga19], insbesondere: Aufgabe, Artefakt, Aufgabentext, Coderahmen, Testtreiber, Musterlösung, Grader, Schablone, Variationspunkt, Platzhalter, Wertebereich und -belegung, Materialisierung.

3 Haskell-Aufgaben in Autotool

Eine konkrete Aufgabe gemäß [SVW19] kann stilisiert bisher etwa wie folgt aussehen:

```
1  configGhcWarnings:
2  - missing-signatures
3  - incomplete-patterns
4  configHlintErrors:
5   - Redundant ==
6  allowRemoving: false
7  ...
8  configLanguageExtensions:
9  - TupleSections
10 ----------
11 module Task01 where
12 import Prelude hiding (and)
13
14 -- Write a function that builds the conjunction of elements in a list.
```

```
15 -- Use structural recursion via foldr.
16 conj :: [Bool] -> Bool
17 conj list = foldr undefined undefined list
18 ----------
19 module Test (test) where
20 import qualified Task01
21 import Test.QuickCheck
22 import ...
23
24 test :: [ Test ]
25 test = [ "compatible with concatenation" ~: qcWithTimeout 500 $ \xs ys ->
26            Task01.conj (xs ++ ys) == (Task01.conj xs && Task01.conj ys),
27          ... ]
```

An Artefakten sind enthalten: in Zeilen 1–9 die Konfiguration für Compiler, Linter und weitere Syntaxüberprüfung, in Zeilen 11–17 Aufgabentext und Coderahmen, in Zeilen 19–27 der Testtreiber. Angezeigt wird den Studierenden nur der mittlere Abschnitt. Eine Musterlösung als weiteres Artefakt wird separat gehalten, nicht in Autotool hinterlegt (jedoch in der Regel im Zuge des Hochladens einer Aufgabe von Lehrendenseite einmalig testweise eingereicht).

Die Verwendung eines reinen Textformats und einfache Einbettung des Aufgabentextes als programmiersprachlicher Kommentar wie in [ROS10], gegenüber einer mit mehr Struktur versehenen Ablage der Artefakte wie in ProFormA [Re19] (die gegebenenfalls auch noch weitere Artefakte wie Bewertungsschema und Binärdateien umfasst), ist neben praktikabler Versionskontrolle wesentlich der Konzeption von Autotool geschuldet. Tatsächlich ist eine Nutzung mit anderen Lernmanagementsystemen oder Gradern bisher nicht vorgesehen.

4 Variantenbildung via Text-Templating

Wie bereits bei mehreren in Abschnitt 2 genannten Arbeiten wird Variabilität durch Textersetzung in Schablonen realisiert. Dafür werden mindestens eine Möglichkeit zur Kennzeichnung von Variationspunkt-Platzhaltern und eine separate Angabe von einzufügenden Werten bzw. Wertebereichen je Platzhalter benötigt. In Anlehnung an übliche Templating- und String-Interpolations-Umsetzungen, in Haskell wie anderen Sprachen, wurde die Syntax `#{platzhalter}` eingeführt. Die Belegung von Platzhaltern erfolgt in einem neuen, der Textdatei vorangestellten Abschnitt. Im einfachsten Fall besteht die Belegung in der Angabe genau eines festen einzusetzenden Strings. Soll etwa von der Aufgabennummer abstrahiert werden, könnte dies für das Beispiel aus Abschnitt 3 so aussehen:

```
taskName = return "Task01"
----------
vormalige Zeilen 1–9
----------
```

```
module #{taskName} where
vormalige Zeilen 12–17
----------
module Test (test) where
import qualified #{taskName}
vormalige Zeilen 21–24
test = [ "compatible with concatenation" ~: qcWithTimeout 500 $ \xs ys ->
           #{taskName}.conj (xs ++ ys) == (#{taskName}.conj xs && ...),
         ... ]
```

Da die angelegte Musterlösung zu Aufgabentext und Coderahmen passend bleiben (im konkreten Fall, den gleichen Modulnamen verwenden) muss, werden Aufgabe und Musterlösung immer gemeinsam verarbeitet, wobei aus der Aufgabe stammende Wertbelegungen – auch im späteren Fall der zufallsbasierten Erzeugung – automatisch auch in der Musterlösungs-Datei sichtbar sind, also dort interpoliert werden können ohne erneute Einführung.

Eine einfache Nutzung wie oben dient noch nicht der etwaigen Variabilität über an verschiedene Studierende desselben Kurses ausgespielte Aufgabentexte und Coderahmen hinweg. Vielmehr wird sie eingesetzt, um die Wiederverwendung von Aufgaben über mehrere Durchführungen des Kurses hinweg zu vereinfachen. Zum Beispiel könnte eine Aufgabe aus einem Aufgabenpool in verschiedenen Jahren genutzt werden, dabei aber eine Anpassung der Aufgabennummer, von Verweisen auf Vorlesungsfolien, angegebenen URLs etc. nötig sein. Allein schon das explizite „Herausziehen“ der relevanten Angaben in einen sofort in den Blick fallenden Präfix der Aufgaben-Textdatei erleichtert hier das Management und vermeidet das Vergessen von Anpassungen.

Nicht nur in den Haskell-Teilen der Textdatei ist die Interpolation von Werten möglich. So könnten etwa auch in dem YAML-Abschnitt zur Tool-Konfiguration „`configGhcWarnings: ...`“ variable Inhalte auftreten. Gebrauch wird hiervon aktuell zum Beispiel dafür gemacht, den im Kontext der im folgenden Abschnitt besprochenen Randomisierung herangezogenen Seed an einer für die Studierenden nicht einsehbaren Stelle ihrer jeweils materialisierten Aufgabe zu persistieren: „`# the seed used was: #seed`“.

5 Randomisierung und deren Steuerung

Teils bereits im Übungsbetrieb, und umso mehr in nur begrenzt überwachbaren Prüfungskontexten, ist neben fixer Ersetzung auch eine zufallsbasierte Gestaltung von Aufgabentext und/oder Coderahmen wünschenswert. Hierzu wird eine Kombination aus der bewährten QuickCheck-Wertegenerator-DSL und eigens entwickelten Funktionen angeboten. Eine Beispielverwendung sah wie folgt aus:

```
1 wordingWatermark = withCurrentSeed (elements ["are still", "remain"])
2 start = withSeed (elements [0, 1]) (#{seed} + 1)
```

```
3  ...
4  ----------
5  module ExamTask02 where
6  ...
7
8  -- Here is a concrete list comprehension:
9  original :: [ (Int, Int) ]
10 original = [ (u,v) | u <- [#{start}..100], v <- [1..u], u `mod` v > 0 ]
11
12 -- Your task is to implement the same without using list
13 -- comprehensions (though range expressions #{wordingWatermark} allowed).
14 ...
```

Hieraus sind vier verschiedene konkrete Aufgaben materialisierbar, indem einerseits eine leichte Umformulierung im Aufgabentext eingesetzt wird, andererseits der Startwert einer im Coderahmen gegebenen Liste zwischen `0` und `1` variiert.[2]

Einige Anmerkungen an dieser Stelle:

1. Der `seed`-Wert wird prinzipiell systemseitig gesetzt, kann jedoch (wie einige andere angebotene Default-Platzhalter und Spezial-Variationspunkte) auch innerhalb des Präfixes der Aufgaben-Textdatei explizit überschrieben werden, was etwa zu Debugging- und Reproduktionszwecken nützlich sein kann.

2. Statt wie im Beispiel nur die `elements`-Primitive (die per probabilistischer Gleichverteilung einen Wert aus einer Liste wählt) kann die gesamte Ausdrucksmächtigkeit von QuickCheck zum Einsatz gebracht werden. Lehrende im Haskell-Kontext werden dazu in der Regel viel Erfahrung mitbringen, etwa aus der Verwendung dieser DSL auch in Testtreibern.

3. Wie im Fall von `start` zu sehen, können nicht nur Strings erzeugt werden, sondern praktisch beliebig getypte Werte. Es muss einzig möglich sein (durch Vorhandensein oder Bereitstellen einer Instanz der Typklasse `Show`), an der Interpolationsstelle eine Umwandlung in Strings vorzunehmen.

4. Die unterscheidende Verwendung von einerseits `withCurrentSeed`, andererseits `withSeed ... (#{seed} + 1)`, dient der „Entkopplung“ der beiden Zufallswahlen. Es soll hier also vermieden werden, dass nur zwei konkrete Aufgabenstellungen entstehen, weil durch zweimalige Verwendung von `withCurrentSeed` die beiden 1-aus-2 Wahlen korrelieren würden, also `wordingWatermark = "are still"` immer zusammen mit `start = 0` und `wordingWatermark = "remain"` immer zusammen mit `start = 1` auftreten würde. Stochastisch sicherere Entkopplungen als Verwendung von[3] `#{seed}` und `#{seed}+1` wären

[2] Da die Korrektheit einer Einreichung dann auch von letzterem Variationspunkt abhängt, muss der Testtreiber-Abschnitt nach Zeile 14 natürlich ebenso auf `#{start}` zugreifen.

[3] Praktisch gilt: `withCurrentSeed (...) = withSeed (...) #{seed}`.

prinzipiell natürlich auch umsetzbar. Gleichermaßen kann es allerdings auch vorkommen, dass eine Korrelation verschiedener zu treffender Auswahlen bewusst gewünscht ist, weil dies etwa zur Variationsstrategie bei einer gewissen Aufgabe passt oder für die Korrektheit der Musterlösung wichtig ist.

Ein Beispiel für den letztgenannten Aspekt ist folgender Auszug aus einer Aufgabe zur Angabe von Werten mit bestimmten vorgegebenen Typen:

```
pairTypes = withCurrentSeed (elements ["Bool, [Int]", "[Int], Bool"])
pairValues = withCurrentSeed (elements ["False, []", "[], False"])
----------
...
value1 :: Either Bool (#{pairTypes})
value1 = undefined

value2 :: Either Bool (#{pairTypes})
value2 = undefined
...
```

und dann in der zugehörigen Musterlösungs-Datei:

```
value1 = Left True
value2 = Right (#{pairValues})
```

Im Geiste des weiter oben angeführten Beispiels mit Variation tatsächlich die Semantik beeinflussender Teile der Aufgabenstellung, jedoch über einfachen Austausch von Zahlenwerten (`start = 0` vs. `start = 1`) hinausgehend, bewegt sich eine ebenfalls in einer Klausur verwendete Aufgabe, welche die Ersetzung allgemeinerer Teilausdrücke demonstriert:[4]

```
predicate = withSeed (elements ["even x", "odd x "]) (#{seed} + 1)
factor = withSeed (elements [2, 3]) (#{seed} + 2)
----------
...

original :: [Int] -> Int
original [] = 1
original (x:xs) | #{predicate}    = x - #{factor} * original xs
                | otherwise = original xs

... (Aufgabenstellung ist die Umschreibung in foldr-Form.)
```

[4] Der Einfachheit halber werden die in den Coderahmen einzusetzenden Teilausdrücke weiterhin ausschließlich als Strings behandelt. Es findet also insbesondere keine Scope-Überprüfung statt, um etwa herauszufinden, ob das `x` in `"even x"` an der letztendlichen Verwendungsstelle überhaupt gebunden sein wird.

Für ein Beispiel, das die Variierungsmöglichkeiten bewusst etwas exzessiv ausreizt, und gleichzeitig die Verwendung von nicht nur QuickCheck-Primitiven sondern der allgemeinen Ausdruckssprache von Haskell in Wertegeneratoren demonstriert, sei auf die Datei `Haskell-07.hs` jeweils in den Unterverzeichnissen `tasks` und `solutions` von `https://github.com/fmidue/haskell-template-parametrization/tree/main/exam-examples` verwiesen. Unter anderem ist dort zu sehen, wie im Präfix der Aufgaben-Textdatei auch selbst wieder Interpolation stattfinden kann, also verschiedene Platzhalter Bezug aufeinander nehmen können.

Aus dem vorigen Absatz, aber auch bereits aus den nummerierten Anmerkungen **2.** und **4.** oben ist ersichtlich, dass wir für eine effektive Anwendung unseres Ansatzes von Lehrenden-als-Haskell-Programmierer ausgehen. Dies schränkt die Übertragbarkeit auf andere Szenarien und erst recht auf andere Programmiersprachen ein, deckt sich jedoch mit unserer Praxis. Wir gewinnen eine hohe Ausdrucksmächtigkeit hinsichtlich der Gestaltung und Kombination von Wertbelegungen, ihren Abhängigkeiten und Constraints, die deutlich über der von etwa [ROS10] liegt. Auch verglichen mit [Ga19] bzw. dem dort verwendeten Datenmodell aus [Ga18] dürfte zumindest die pragmatische Ausdrucksmächtigkeit höher sein. Zwar könnten dort über Javascript-Ableitungen prinzipiell auch beliebig komplexe Werteverteilungen samt Abhängigkeiten etc. realisiert werden, jedoch ohne die Vorteile einer genau für diesen Zweck geschaffenen und bewährten DSL wie QuickCheck. Was [Ga18] mitbringt und bei uns fehlt, ist die Möglichkeit der händischen Auswahl von Wertbelegungen über einen modellbasiert generierten GUI-Dialog.

6 Einbringen von Wasserzeichen

Bereits in einem Beispiel zu Beginn von Abschnitt 5 eingeführt wurde die Verwendung geringer, und nicht auf den ersten Blick auffälliger Umformulierungen des Aufgabentextes. Gedacht war dies als niedrigschwellige Maßnahme zur Detektion von Plagiaten. Obwohl keinen Einfluss auf die Aufgabensemantik habend, und obwohl solche Kommentartexte vor Einreichung von Lösungen sogar gelöscht werden dürfen, können besonders aufwandsarme Übernahmen von fremden Lösungen so bereits auffällig werden.

Ein Weg, dies auf die Spitze zu treiben, ist die Kombination vieler lokaler Änderungen, um geeignet exponentiell viele Aufgabentexte zu erhalten. Ein anderes Experiment, das durchgeführt wurde, war die Verwendung noch mehr versteckter Wasserzeichen, innerhalb von Whitespace in der Aufgabenstellung. Dazu wurde ein Spezial-Variationspunkt eingeführt, der im Präfix gesetzt werden kann: „`enableWhitespaceWatermarking = return "True"`“, und auf den dann zwar nicht weiter Bezug genommen wird, der aber systemseitig eine nach gewissen wiederkehrenden Mustern fast zufällige Hinzufügung von Leerzeichen an Zeilenenden der an die Studierenden auszuspielenden Dateien bewirkt. Während die Aufgabenstellungen dann visuell selbst nicht mehr unterscheidbar sind, kann mit geeignetem Diff-Tool oder Sichtbarmachung in Editoren o.ä. das Kopieren von Dateien zwischen

Studierenden (die in der Klausursituation eigentlich nicht hätten kommunizieren dürfen) aufgespürt werden, siehe Abb. 1.

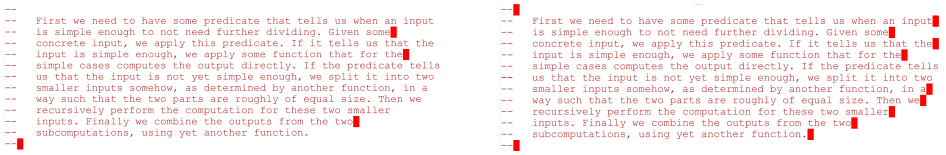


Abb. 1: Auszüge von aus gleicher Schablone materialisierten Aufgabentexten, einzig durch „normalerweise unsichtbare“ Leerzeichen verschieden voneinander.

Literaturverzeichnis

[CH00] Claessen, K.; Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: Proc. ICFP, ACM, S. 268–279, 2000.

[CM19] Combéfis, S.; de Moffarts, G.: Automated Generation of Computer Graded Unit Testing-Based Programming Assessments for Education. In: Proc. CSEIT, CoRR, S. 91–100, 2019.

[Ga18] Garmann, R.: Ein Schnittstellen-Datenmodell der Variabilität in automatisch bewerteten Programmieraufgaben. In: Proc. SEELS, CEUR, S. 52–56, 2018.

[Ga19] Garmann, R.: Ein Datenformat zur Materialisierung variabler Programmieraufgaben. In: Proc. ABP, GI, S. 51–58, 2019.

[Ko22] Koth, L.: Parametrisierte Konfiguration von Haskell-Übungsaufgaben, Bachelor thesis, University of Duisburg-Essen, 2022.

[OG17] Otto, B.; Goedicke, M.: Auf dem Weg zu variablen Programmieraufgaben: Requirements Engineering anhand didaktischer Aspekte. In: Proc. ABP, CEUR, 2017.

[Re19] Reiser, P.; Borm, K.; Feldschnieders, D.; Garmann, R.; Ludwig, E.; Müller, O.; Priss, U.: ProFormA 2.0 – ein Austauschformat für automatisiert bewertete Programmieraufgaben und für deren Einreichungen und Feedback. In: Proc. ABP, GI, S. 43–50, 2019.

[ROS10] Radošević, D.; Orehovački, T.; Stapić, Z.: Automatic On-line Generation of Student's Exercises in Teaching Programming. In: Proc. CECIIS, SSRN, 2010.

[Sa22] Sarsa, S.; Denny, P.; Hellas, A.; Leinonen, J.: Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In: Proc. ICER, ACM, S. 27–43, 2022.

[So21] Sovietov, P.: Automatic Generation of Programming Exercises. In: Proc. TELE, IEEE, S. 111–114, 2021.

[SS22] Strickroth, S.; Striewe, M.: Building a Corpus of Task-Based Grading and Feedback Systems for Learning and Teaching Programming. International Journal of Engineering Pedagogy 12/5, S. 26–41, 2022.

[SVW19] Siegburg, M.; Voigtländer, J.; Westphal, O.: Automatische Bewertung von Haskell-Programmieraufgaben. In: Proc. ABP, GI, S. 19–26, 2019.

[Wa17] Waldmann, J.: Automatische Erzeugung und Bewertung von Aufgaben zu Algorithmen und Datenstrukturen. In: Proc. ABP, CEUR, 2017.

[We20] Westphal, O.: A Framework for Generating Diverse Haskell-I/O Exercise Tasks. In: Proc. WFLP, Springer, S. 97–114, 2020.

Fehlvorstellungen in der Programmierausbildung: Eine Heuristik für die semi-automatische Annotation von Fehlerkandidaten

Björn Fischer[1], Sven Eric Panitz[2] und Ralf Dörner[3]

Abstract: Die zuverlässige Erkennung von Fehlern zu Fehlvorstellungen in der Programmierausbildung stellt eine Herausforderung dar, die mit Deep Learning adressiert werden kann. In dieser Arbeit wird eine Heuristik vorgestellt, die es ermöglicht, die dafür erforderlichen Annotationen weitestgehend automatisch zu generieren. Die Heuristik verbindet Informationen aus der statischen und dynamischen Codeanalyse mit dem Ziel, mögliche Fehlalarme zu reduzieren. Erste Ergebnisse zeigen in unserem Datenfall anhand eines betrachteten Fehlertyps, dass die Heuristik in etwa der Hälfte der Fälle eine automatische Entscheidung treffen kann und dabei eine Genauigkeit von 81 % erreicht. Dies stellt eine erhebliche Verbesserung von etwa einem Drittel gegenüber den Ergebnissen von Pattern Matching dar.

Keywords: Introductory Programming; Intelligent Tutoring Systems; Automated Feedback; Misconceptions; Bytecode Instrumentation; Code Coverage; Machine Learning; Weak Supervision

1 Einführung

Die automatische Generierung von Feedback zu Fehlvorstellungen in der Programmierausbildung könnte eine Maßnahme zur Adressierung der hohen Abbrecherquote in der Informatik darstellen. Die Definition der *Fehlvorstellung* wird in der Forschung umfangreich und kontrovers diskutiert. Qian und Lehman [QL17] setzen sich mit unterschiedlichen Definitionen auseinander und kommen zu dem Ergebnis, dass Fehlvorstellungen – aufgrund der unterschiedlichen Auslegungen des Begriffs – zusammenfassend als Fehler im konzeptionellen Verständnis von Programmierkonzepten bezeichnet werden können. Im Rahmen dieser Arbeit stehen vor allem die Indikatoren im Vordergrund, die auf mögliche Fehlvorstellungen hinweisen können und damit für eine automatische Generierung von Feedback näher betrachtet werden müssen. Zu den Indikatoren zählen Programmfehler, die im weiteren Verlauf der Arbeit als Fehler bezeichnet und in Abschnitt 2 näher definiert werden. Die Generierung von Feedback setzt eine zuverlässige Erkennung von Fehlern voraus, da eine hohe Anzahl von Fehlalarmen zu einer Ablehnung oder Nichtbeachtung des Feedbacks führen könnte. Für die Erkennung können Pattern Matching Algorithmen verwendet werden. Diese bringen jedoch die Herausforderung mit sich, mögliche Fehler von tatsächlichen Fehlern zu unterscheiden. Ein Lösungsansatz für die Unterscheidung zwischen möglichen und tatsächlichen Fehlern besteht in der Verwendung von Deep Learning. Mögliche Fehler bezeichnen wir hierbei als *Fehlerkandidaten*. Das Training von

[1] Hochschule RheinMain, Fachbereich DCSM, bjoern.fischer@hs-rm.de
[2] Hochschule RheinMain, Fachbereich DCSM, sveneric.panitz@hs-rm.de
[3] Hochschule RheinMain, Fachbereich DCSM, ralf.doerner@hs-rm.de

doi:10.18420/abp2023-8

Deep Learning Modellen erfordert Annotationen zu diesen Fehlerkandidaten im Quellcode. Annotationen sind Klassifizierungsinformationen, die für das Training von Deep Learning Modellen benötigt werden. Aktuelle Ansätze, wie der von Shi et al. [Sh21], erfordern eine manuelle Erzeugung dieser Annotationen. Soweit wir wissen, untersucht keine Arbeit, wie Testfälle dazu verwendet werden können, um den Aufwand für die Annotation zu verringern. Der Beitrag dieser Arbeit besteht daher aus der Entwicklung einer Heuristik, die Informationen aus der statischen und dynamischen Codeanalyse verbindet, um Annotationen zu Fehlerkandidaten zuverlässig und weitestgehend automatisiert zu erzeugen. Wir adressieren damit die folgende Forschungsfrage: *Können Testfälle zu Programmieraufgaben dazu verwendet werden, um Fehlalarme von Pattern Matching bei der Erkennung von Fehlern zu Fehlvorstellungen zu reduzieren?* Fehlvorstellungen selbst werden nicht näher betrachtet, da eine adäquate Einordnung in dieser Arbeit aufgrund der Komplexität nicht erfolgen kann. Die Beantwortung der Forschungsfrage erfolgt eingeschränkt anhand von Fehlern des Typs *UnusedReturnValue*, bei denen Rückgabewerte von Methoden nicht verwendet werden, obwohl dies erforderlich wäre. Dabei handelt es sich um *einen* möglichen Fehlertyp zu *einer* Fehlvorstellung. Diese betrifft die Funktionsweise von Rückgabewerten und wird nach Qian und Lehman unter *Difficulties in Conceptual Knowledge* eingeordnet. Der Fehlertyp *UnusedReturnValue* wird in der Studie von Brown und Altadmri [BA17] als einer der häufigsten und am schwierigsten zu behebenden Fehlertypen beschrieben. Daraus geht jedoch nicht hervor, in wie vielen Fällen es sich um einen tatsächlichen Fehler handelte.

2 Erkennung von Fehlern zu Fehlvorstellungen

Nach Gusukuma et al. [Gu18] können sich Fehlvorstellungen über Fehler äußern, die als *fehlerhafte* Zusammenstellung von Sprachkonstrukten im Quellcode definiert werden. Es gibt jedoch auch solche Zusammenstellungen, die keine Auswirkungen auf die funktionale Korrektheit haben und trotzdem Indikatoren für Fehlvorstellungen darstellen. Dazu zählen unnötige Vergleiche mit boolschen Literalen. Zur Abgrenzung definieren wir daher den Begriff *Symptom* als Oberbegriff für Zusammenstellungen von Sprachkonstrukten im Quellcode, die aus Fehlvorstellungen folgen können. *Fehler* definieren wir als Symptome, die eine falsche Anwendung von Programmierkonzepten verkörpern und sich damit auf die funktionale Korrektheit auswirken *können*. Es kann dazu kommen, dass ein Fehler die funktionale Korrektheit nicht beeinflusst, wenn die Fehlerwirkung durch Folgeanweisungen wieder aufgelöst wird. Ferner führt eine falsche Problemlösestrategie nicht zwangsläufig zu einem Fehler, da Programmierkonzepte trotzdem korrekt eingesetzt werden können. Für die Erkennung von Fehlern können Werkzeuge für die statische Codeanalyse, wie ErrorProne [Go23], eingesetzt werden. Diese arbeiten mit Pattern Matching, um Fehlerkandidaten im Quellcode zu erkennen. Im professionellen Umfeld wurden jedoch bereits hohe Fehlalarmraten von 35 % bis 91 % bei der Verwendung von Pattern Matching beobachtet [MS23]. Da sich Fehler in der Programmierausbildung von denen unterscheiden, die in einem professionellen Umfeld auftreten, entwickelten Gusukuma et al. ein Verfahren für die Beschreibung und Erkennung von Fehlerkandidaten im Kontext von Fehlvorstellungen.

Es ist fraglich, ob Pattern Matching in der Programmierausbildung bessere Ergebnisse erzielen würde. Um Fehlalarme zu reduzieren, wurden zahlreiche Ansätze entwickelt, die allerdings nur den Einsatz dieser Werkzeuge in einem professionellen Umfeld adressieren [MS23]. Darüber hinaus untersuchten Shi et al. [Sh21], ob Fehler in annotierten Lösungen durch Deep Learning Modelle erkannt werden können. Off-by-one Fehler wurden dabei nur in der Hälfte der Fälle korrekt erkannt. Diese Fehler entstehen beispielsweise durch die falsche Verwendung von Zählervariablen und hängen stark von Kontextinformationen ab. *Kontextinformationen* definieren wir als Merkmale in umliegendem Quellcode, die für eine korrekte Entscheidung einbezogen werden müssen. Es ist unklar, ob mehr annotierte Lösungen zu einer Verbesserung führen würden. Einer Überprüfung dieser Hypothese steht jedoch eine Limitierung des Ansatzes entgegen: Für die manuelle Annotation fällt ein erheblicher Arbeitsaufwand an. Die Annotation von etwa 1.800 fehlerhaften Lösungen mit drei Fehlertypen benötigte in der Arbeit von Shi et al. bereits 20 Stunden. Haldeman et al. [Ha21] zeigten eine Perspektive hierfür auf, indem Testfälle in die Erkennung einbezogen wurden. Dabei wurde erste Evidenz für die Hypothese gefunden, dass Lösungen, die an den selben Testfällen scheiterten, auch die selben Fehler aufweisen. Der Ansatz verwendet jedoch kein Deep Learning und basiert ebenfalls auf einem manuellen und zeitaufwändigen Verfahren. Diese Nachteile werden in der vorliegenden Arbeit adressiert.

3 Heuristik

Die Heuristik ist Teil einer Methode im Bereich des maschinellen Lernens, die das Ziel verfolgt, Kontextinformationen von Fehlern über ein Deep Learning Modell zu lernen. Das Training des Deep Learning Modells erfolgt dabei überwacht mit Annotationen, die durch die Heuristik automatisiert erzeugt werden. Die der Heuristik zu Grunde liegende Hypothese besteht darin, dass insbesondere korrekte Lösungen dazu verwendet werden können, um Fehlalarme aus dem Pattern Matching zu identifizieren. Bei der Methode handelt es sich um eine Form von Weak Supervision, da die erzeugten Annotationen fehlerbehaftet sein können.

Erkennung von Fehlerkandidaten. Lösungen werden zunächst mit der Bibliothek JavaParser [SVT21] in einen Abstract Syntax Tree (AST) geparsed. Für den Fehlertyp *UnusedReturnValue* wurde ein Detektor entwickelt, der Methodenaufrufe als Knoten im AST zurückgibt. Der Detektor entspricht der Implementierung in ErrorProne und anderen Werkzeugen, indem er einen Fehlerkandidaten für jeden Methodenaufruf ermittelt, wenn a) der Rückgabewert nicht verwendet wird und b) die aufgerufene Methode keinen *trivialen* Seiteneffekt hat. *Triviale Seiteneffekte* definieren wir als Seiteneffekte, die durch fest hinterlegte Signaturen von Methoden in der Java Standardbibliothek bestimmt werden. Eine Methode hat nach Sălcianu und Rinard [SR05] einen Seiteneffekt, wenn der vorherige Programmzustand durch den Methodenaufruf verändert wird. Da I/O Operation es rechtfertigen können, den Rückgabewert zu ignorieren, ergänzen wir diese Definition so, dass Methoden mit I/O Operationen ebenfalls einen Seiteneffekt haben. Abbildung 1 verdeutlicht, dass

```
public LL<A> add(A a) {
    if (this.isEmpty()) {
        return new LL<>(a, new LL<>());
    } else {
        this.tl.add(a);
    }
    return this;
}
```

```
public LL<A> add(A a) {
    if (this.isEmpty()) {
        this.hd = a;
        this.tl = new LL<>();
    } else {
        this.tl.add(a);
    }
    return this;
}
```

Abb. 1: Auszug aus zwei Java-Lösungen zu einer LinkedList-Implementierung, die einen tatsächlichen Fehler (links) und einen falsch erkannten Fehler (rechts) zeigen. Rot hinterlegt sind die Fehlerkandidaten. Gelb hinterlegt sind die Kontextinformationen, die zur Entscheidung führen: Eine Missachtung des Rückgabewerts ist nur in der rechten Lösung zulässig, da dort ein Seiteneffekt vorliegt.

der Detektor nicht ausreicht, da es zu Fehlalarmen kommen kann. Das Ziel der Heuristik ist es daher, die Fehlerkandidaten zu bewerten. Einen Fehlerkandidaten zum Fehlertyp *UnusedReturnValue* definieren wir als Fehler, wenn der Rückgabewert nicht verwendet wird und die aufgerufene Methode keinen Seiteneffekt aufweist.

Instrumentierung. Da die Bewertung der Fehlerkandidaten von den Testergebnissen abhängt, werden für jede Lösung die Testfälle der zugehörigen Aufgabe ausgeführt. Dabei wird *für jeden* Testfall gespeichert, ob dieser a) erfolgreich war und b) welche Zeilen in der Lösung durchlaufen wurden. Hierfür wird der Bytecode der Lösung mit JaCoCo [Ho23] instrumentiert. Bei der Bytecode-Instrumentierung werden Marker nachträglich in ein Programm eingefügt, um Informationen über das Laufzeitverhalten zu sammeln [Te16].

Generierung der Annotationen. Die Heuristik nutzt die erkannten Fehlerkandidaten und Informationen aus der Instrumentierung, um die Annotationen zu generieren. Bei dem Entwurf der Heuristik wurden Java-Lösungen untersucht, die in einem Automated Assessment System (siehe Abschnitt 4) erhoben wurde. Bei den Lösungen handelt es sich um komplexe Klassen mit mehreren Methoden, die teilweise abhängig voneinander getestet werden. Da Fehler nicht immer Auswirkungen auf die gesamte Lösung haben, könnte eine Betrachtung der Gesamtheit aller Testfälle zu falschen Aussagen führen. Weiterhin sollte eine Aussage nur zu erreichbarem Code getroffen werden, da Fehler in nicht erreichbarem Code keinen Einfluss auf die Korrektheit haben. Daher werden nur die relevanten Testfälle betrachtet, die anhand der Instrumentierung abgeleitet werden. Einen Testfall $t \in T$ definieren wir in Hinsicht auf einen Fehlerkandidaten $c \in C$ als *relevant*, wenn es eine Zeile im Bereich von c gibt, die von t ausgeführt wurde. Zeilen gelten als ausgeführt, wenn mindestens eine Instruktion darin ausgeführt wurde. Die Menge aller relevanten Testfälle in Hinsicht auf c bezeichnen wir als T_c und die erfolgreichen relevanten Testfälle als $T_c^1 \subseteq T_c$. Anschließend definieren wir die Wahrscheinlichkeit für $ist\_fehler(c)$ als:

$$P(ist\_fehler(c)) = 1 - \frac{|T_c^1|}{|T_c|} \text{ mit } |T_c| \neq 0$$

Die Heuristik $h : C \rightarrow \{true, false, abstain\}$ wird in (1) dargestellt. Bei $true$ oder $false$ wird der Wert als Annotation übernommen. Bei $abstain$ konnte aufgrund von fehlenden oder unklaren Testergebnissen keine Entscheidung getroffen werden. Die Parameter k_{true} und k_{false} stellen Schwellenwerte dar, die im Rahmen der Optimierung festgelegt werden.

$$h(c) = \begin{cases} true & \text{wenn } |T_c| \neq 0 \text{ und } P(ist\_fehler(c)) > k_{true} \\ false & \text{wenn } |T_c| \neq 0 \text{ und } P(ist\_fehler(c)) \leq k_{false} \\ abstain & \text{sonst} \end{cases} \quad (1)$$

4 Experiment

Zur Beantwortung der Forschungsfrage wird die vorgestellte Heuristik den Ergebnissen aus dem Pattern Matching gegenübergestellt, die durch den *UnusedReturnValue*-Detektor erzeugt werden. Die Evaluation erfolgt jeweils über den Abgleich mit manuell annotierten Fehlerkandidaten, welche die Ground Truth darstellen. Als Datengrundlage verwenden wir einen Teil des Subato-Datensatzes, der aus dem Lehrbetrieb des Automated Assessment Systems Subato[4] der Hochschule RheinMain entstanden ist. Im Gegensatz zu öffentlichen Datensätzen, wie CodeWorkout[5] und CodeNet[6], enthält dieser Datensatz den Quellcode von Testfällen, der für die Instrumentierung notwendig ist. Zudem bestehen die Lösungen aus komplexen Klassen und nicht nur aus kleinen Methoden, wie in CodeWorkout. Der relevante Teil des Datensatzes besteht aus etwa 100.000 Java Lösungen, die von Informatikstudierenden in einführenden Programmierveranstaltungen zwischen 2017 und 2022 zu bewerteten Aufgaben eingereicht wurden. Zwei Lehrende annotierten 892 Fehlerkandidaten des Fehlertyps *UnusedReturnValue* aus 532 Lösungen. Dabei wurden nur Lösungen annotiert, zu denen Testergebnisse ermittelt werden konnten. Zudem wurden Fehlerkandidaten bei der Annotation exkludiert, die sich in einer Methode befanden, deren Signatur einer typischen Main-Methode entspricht und die nicht durch einen Testfall aufgerufen wurden. In diesen Fällen kann keine Entscheidung getroffen werden, da die Intention des Autors nicht festgestellt werden kann. Die Anwendung von Cohen's Kappa ergab einen κ-Wert von 0,87. Anschließend wurden die Lösungen in einen Validierungs- und Testdatensatz aufgeteilt. Der Validierungsdatensatz dient zur Optimierung der Parameter aus der Heuristik h (1). Daher erfolgte die Aufteilung so, dass a) sich alle Lösungen eines Studierenden für

[4] https://subato.cs.hs-rm.de/

[5] https://sites.google.com/ncsu.edu/csedm-dc-2021/dataset

[6] https://github.com/IBM/Project_CodeNet

die selbe Aufgabe entweder im Validierungs- oder Testdatensatz befinden, b) das Verhältnis der Größe von Validierung- und Testdatensatz ungefähr 70 % : 30 % beträgt und c) die beiden Klassen ungefähr balanciert sind. Im Validierungsdatensatz befanden sich dann 311 Lösungen (519 Fehlerkandidaten, davon 258 Fehler), im Testdatensatz 221 Lösungen (373 Fehlerkandidaten, davon 184 Fehler).

5 Ergebnisse

Die Optimierung der Parameter auf dem Validierungsdatensatz ergab, dass $k_{true} = 0,9$ und $k_{false} = 0,2$ die besten Ergebnisse in Hinsicht auf die Genauigkeit erzielte (Precision: 0,92 - Recall: 0,75 - F1: 0,82). Von den 519 Fehlerkandidaten im Validierungsdatensatz hat die Heuristik in 218 Fällen (42 %) eine Entscheidung getroffen. Zu den übrigen Fehlerkandidaten konnten keine relevanten Testfälle ermittelt werden oder die Wahrscheinlichkeit lag zwischen den Schranken. Anschließend wurde die Heuristik mit den optimierten Parametern auf dem Testdatensatz evaluiert. Von den 373 Fehlerkandidaten im Testdatensatz hat die Heuristik in 200 Fällen (53,6 %) eine Entscheidung getroffen. Tabelle 1 zeigt die Ergebnisse der Heuristik auf dem Testdatensatz im Vergleich zur Baseline, welche die Pattern Matching Ergebnisse darstellt. Daraus wird ersichtlich, dass die Heuristik in Hinsicht auf die Genauigkeit um etwa ein Drittel besser abschneidet als die Baseline.

	Precision	Recall	F1	N
Heuristik	**0,81**	**0,81**	**0,81**	**200**
Baseline	0,48	1,00	0,65	200
Baseline (alle)	0,49	1,00	0,66	373

Tab. 1: Ergebnisse der Heuristik im Vergleich zur Baseline auf dem Testdatensatz. *Baseline (alle)* umfasst auch die Fehlerkandidaten, zu denen die Heuristik keine Entscheidung getroffen hat.

Abbildung 2 zeigt die Ergebnisse der Heuristik. Diese verdeutlicht, dass Fehlerkandidaten mit hoher Zuverlässigkeit automatisch annotiert werden können, wenn die Testergebnisse der relevanten Testfälle eine deutliche Tendenz aufweisen. Eine Überprüfung der Fehlerkandidaten, zu denen die Heuristik keine Entscheidung getroffen hat, ergab, dass Methoden als nicht ausgeführt gekennzeichnet wurden, obwohl diese ausgeführt wurden. Dies war der Fall, wenn in den Lösungen implizite Exceptions (wie eine `NullPointerException`) aufgetreten sind und der Testfall daher abgebrochen wurde. Dies deckt sich mit der Studie von Tengeri et al. [Te16], die zeigt, dass Bytecode-Instrumentierung zu Verzerrungen bei der Zuordnung zum ursprünglichen Quellcode führen kann.

6 Diskussion und Ausblick

Die Ergebnisse bestätigen eine hohe Anzahl von Fehlalarmen bei der Erkennung von Fehlerkandidaten zum Fehlertyp *UnusedReturnValue* mit Pattern Matching. Unsere Heuristik

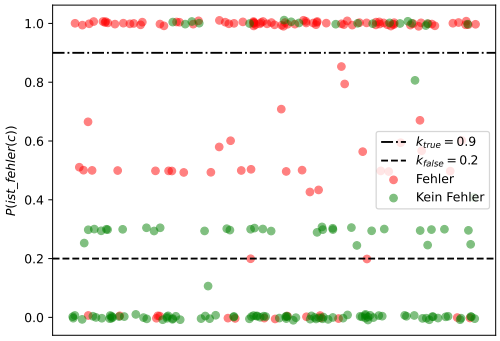


Abb. 2: Ergebnisse der Heuristik auf dem Testdatensatz. Die Datenpunkte stellen die Fehlerkandidaten dar, die farbliche Einteilung entspricht der manuellen Annotation. Die Überlappung der Datenpunkte wurde anhand von Jittering über $\mathcal{N}(0, 0.005^2)$ entlang der y-Achse zur besseren Sichtbarkeit reduziert.

konnte mit hoher Zuverlässigkeit entscheiden, ob es sich bei diesen Fehlerkandidaten um tatsächliche Fehler handelt. Dabei übertraf die Heuristik die Genauigkeit von Pattern Matching in unserem Datenfall deutlich um etwa ein Drittel. Eine automatische Entscheidung konnte jedoch nur bei etwa der Hälfte der Fehlerkandidaten getroffen werden. Insgesamt gibt es erste Evidenz dafür, dass eine Einbeziehung von Testfällen die hohe Anzahl an Fehlalarmen von Pattern Matching reduzieren kann. Es ist jedoch unklar, ob diese Ergebnisse auf andere Fehlertypen verallgemeinert werden können. Zudem wurden relevante Testfälle aufgrund von Ungenauigkeiten in der Bytecode-Instrumentierung mit JaCoCo nicht identifiziert. Während solche Ungenauigkeiten in einem professionellem Umfeld weniger stark ins Gewicht fallen, sind sie in der Domäne der Programmierausbildung besonders relevant, da mit einer hohen Anzahl an impliziten Exceptions zu rechnen ist. Eine weitere Limitierung besteht darin, dass die verwendeten Lösungen nicht repräsentativ genug sein könnten: Die Abgabe von Lösungen erfolgte im Rahmen von bewerteten Studienleistungen und wurde durch eine maximale Anzahl von Versuchen beschränkt. Es ist denkbar, dass Lösungen eher dann eingereicht werden, wenn diese lokal korrekt funktionieren. Zuletzt könnte unsere Aufteilung der Lösungen in den Validierungs- und Testdatensatz zu Verzerrungen geführt haben, da wir eine ähnliche Methodik zu Shi et al. [Sh21] verwendet haben und die Autoren dies ebenfalls angemerkt haben. Eine geeignetere Methodik für die Aufteilung ist uns nicht bekannt und sollte zukünftig erschlossen werden. Weiterhin sollten zukünftige Artbeiten untersuchen, wie zuverlässig andere Fehlertypen mit dieser Heuristik erkannt werden können. Eine weitere Möglichkeit besteht in der Erforschung von Algorithmen zur Erkennung von Seiteneffekten, die bei studentischen Lösungen ausreichend gut funktionieren könnten. Zudem sollten alternative Verfahren zur Instrumentierung evaluiert werden, um die Messfehler zu adressieren. Die Herausforderung besteht darin, dass Werkzeuge für alternative Verfahren, wie beispielsweise Quellcode-Instrumentierung, stark unterrepräsentiert und

veraltet zu sein scheinen. Die vorgestellte Heuristik kann den Aufwand für die Annotation von Fehlerkandidaten reduzieren. Die generierten Annotationen bilden die Grundlage für das Training eines Deep Learning Modells, welches dazu eingesetzt werden kann, um Programmieranfänger besser unterstützen zu können.

Literaturverzeichnis

[BA17] Brown, N. C. C.; Altadmri, A.: Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs. ACM Transactions on Computing Education 17/2, S. 1–21, Juni 2017.

[Go23] Google: Error Prone - Google, 2023, URL: https://errorprone.info/, Stand: 26. 04. 2023.

[Gu18] Gusukuma, L.; Bart, A. C.; Kafura, D.; Ernst, J.: Misconception-Driven Feedback: Results from an Experimental Study. In: Proceedings of the 2018 ACM Conference on International Computing Education Research. ACM, Espoo Finland, S. 160–168, Aug. 2018.

[Ha21] Haldeman, G.; Babeş-Vroman, M.; Tjang, A.; Nguyen, T. D.: CSF: Formative Feedback in Autograding. ACM Transactions on Computing Education 21/3, S. 1–30, Sep. 2021.

[Ho23] Hoffmann, M. R.; Mandrikov, E.; Friedenhagen, M.; Janiczak, B.; Liba, R.; Beck, C.: JaCoCo Java Code Coverage Library, 2023, URL: https://www.jacoco.org/jacoco/, Stand: 10. 05. 2023.

[MS23] Muske, T.; Serebrenik, A.: Survey of Approaches for Postprocessing of Static Analysis Alarms. ACM Computing Surveys 55/3, S. 1–39, März 2023.

[QL17] Qian, Y.; Lehman, J.: Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. ACM Transactions on Computing Education 18/1, S. 1–24, Dez. 2017.

[Sh21] Shi, Y.; Mao, Y.; Barnes, T.; Chi, M.; Price, T. W.: More With Less: Exploring How to Use Deep Learning Effectively through Semi-supervised Learning for Automatic Bug Detection in Student Code. In: Proceedings of the 14th International Conference on Educational Data Mining. S. 8, 2021.

[SR05] Sălcianu, A.; Rinard, M.: Purity and Side Effect Analysis for Java Programs. In: Verification, Model Checking, and Abstract Interpretation. Bd. 3385, Springer Berlin Heidelberg, Berlin, Heidelberg, S. 199–215, 2005.

[SVT21] Smith, N.; Van Bruggen, D.; Tomassetti, F.: Javaparser: Visited. Leanpub, oct. de, 2021.

[Te16] Tengeri, D.; Horváth, F.; Beszédes, Á.; Gergely, T.; Gyimóthy, T.: Negative Effects of Bytecode Instrumentation on Java Source Code Coverage. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). Bd. 1, S. 225–235, März 2016.

Erfahrungen bei der Integration des Autograding-Systems CodeOcean in die universitäre Programmierausbildung

Peter Amthor[1], Ulf Döring[1], Daniel Fischer[1], Jonas Genath[1] und Gunther Kreuzberger[1]

Abstract: Eine effektive und effiziente universitäre Programmierausbildung erfordert zunehmend den Einsatz automatisierter Bewertungssysteme. Im Rahmen des Projekts examING[2] erprobt das Teilprojekt AutoPING den Einsatz des quelloffenen Autograding-Systems CodeOcean für übergreifende Lehrangebote und Prüfungen an der TU Ilmenau mit dem Ziel, selbstgesteuertes und kompetenzorientiertes Lernen zu ermöglichen und zu fördern. Der Beitrag gibt einen Überblick über erste Projekterfahrungen bei der Adaption didaktischer Szenarien in der Programmierausbildung hin zu testgetriebener Softwareentwicklung sowie der Generierung von Feedback. Es werden wesentliche Erkenntnisse aus Sicht der Studierenden und Lehrenden erörtert, Herausforderungen und Lösungsansätze zur Integration und Erweiterung von CodeOcean für neue Anwendungsfelder diskutiert sowie zukünftige Perspektiven eröffnet.

Keywords: Autograder; Programmieren; automatisches Feedback; CodeOcean

1 Einleitung

Das Vermitteln von Programmiersprachen, das Verstehen von Algorithmen und das Erlernen von Entwicklungsmethoden erfordert Zeit, Engagement und intensive Übung. Möglichst direktes und qualitativ hochwertiges Feedback ist für Lernende hierbei von großer Bedeutung. Dieses kontinuierlich und zum richtigen Zeitpunkt bereitzustellen, stellt noch immer eine Herausforderung dar [Je22]. Mit testgetriebener Softwareentwicklung kann Feedback automatisch generiert werden. Eines dieser Systeme ist das Autograding-System CodeOcean. Dank seiner Flexibilität und Skalierbarkeit lässt es sich disziplinübergreifend einsetzen und fördert Qualitätssicherung sowie allseitigen Wissensaustausch in der Programmierausbildung.

In diesem Beitrag geben wir einen Einblick in unsere Erfahrungen bei der Umstellung didaktischer Szenarien in der Programmierausbildung auf die testgetriebene Softwareentwicklung und stellen den dafür umgesetzten, disziplin- und themenübergreifenden Einsatz von CodeOcean in der digitalen Hochschullehre vor. Anhand von vier Anwendungsfällen aus verschiedenen Lehrangeboten der TU Ilmenau erörtern wir wesentliche Erkenntnisse, beschreiben Herausforderungen und diskutieren Lösungsansätze.

[1] Technische Universität Ilmenau, PF 100565, 98684 Ilmenau, Deutschland
{peter.amthor,ulf.doering,daniel.fischer,jonas.genath,gunther.kreuzberger}@tu-ilmenau.de

[2] gefördert durch die Stiftung Innovation in der Hochschullehre

doi:10.18420/abp2023-9

2 CodeOcean als einheitliche Plattform

Angesichts steigender Anforderungen an die Qualität der Hochschullehre und den Einsatz digitaler Bildungstechnologien suchten wir zu Beginn des Projekts nach einer Alternative zur bislang heterogenen Landschaft digitaler Werkzeuge in Programmierkursen an der TU Ilmenau. Ausgehend von unseren langjährigen, durchaus gemischten Erfahrungen zum Einsatz von Moodle-Tests, dem Moodle-Plugin *Virtual Programming Lab* [RRH12] sowie mehreren Eigenentwicklungen wollten wir eine im Produktivbetrieb bewährte Lösung zur automatischen Bewertung von Programmieraufgaben und Quelltext finden, die sich in das hochschulweit eingesetzte Lernmanagement-System (LMS) Moodle[3] integrieren lässt, leicht auf neue Anwendungsszenarien adaptierbar ist und so kursübergreifend sowie studiengangsübergreifend etabliert werden kann. Im Ergebnis einer Recherche (vgl. [SS22]) wählten wir das Autograding-System CodeOcean zur intensiven Erprobung aus.

Die Open-Source-Software CodeOcean[4] wurde vom Hasso-Plattner-Instituts (HPI) für die Durchführung von Programmierübungen in Massive Open Online Courses entwickelt. Das Autograding-System bietet nicht nur zahlreiche Funktionen zur automatischen Bewertung von Programmieraufgaben, sondern stellt auch eine umfassende, webbasierte Ausführungs- und Entwicklungsumgebung mit vielfältigen Protokollierungs-, Hilfs- und Kollaborationsfunktionen zur Verfügung. Durch die Implementierung der Learning-Tools-Interoperability-Schnittstelle kann es mit LMS wie z. B. Moodle kommunizieren. Diese Funktionen ermöglichen es, den gesamten Lehr- und Bewertungsprozess bei der Durchführung von Übungen und Prüfungen für verschiedene Programmiersprachen in einer bestehenden IT-Infrastruktur zu unterstützen. Das HPI verwendet das System für über 60 000 Nutzer und hat damit die Skalierbarkeit in der Praxis nachgewiesen [Se21].

CodeOcean dient als Frontend für die Studierenden und organisiert deren Einreichungen. Docker[5]-Container weisen den Nutzern individuelle, je nach Programmiersprache und Aufgabenstellung verschiedene, spezialisierte Ausführungsumgebungen (Kompilier- und Ausführungstools, Bibliotheken, etc.) zu. Indem jedem Nutzer temporär ein separater Container zugewiesen wird, können die Einreichungen untereinander nicht interferieren und es wird keine spezielle Hard- oder Software für Studierende vorausgesetzt [Se21].

3 Python-Programmierübungsreihe für Studienanfänger

Erste Erfahrungen mit CodeOcean sammelten wir in einem Erstsemesterkurs für Studierende aus verschiedenen Studiengängen. Ziel in diesem Kurs ist es, Studierenden mit wenig bis keiner Programmiererfahrung erste Programmierfähigkeiten zu vermitteln. Hierfür entwickelten wir eine Python-Programmierübungsreihe und setzten diese in CodeOcean um. Der

[3] siehe: `https://moodle.org`
[4] Quellcode auf GitHub: `https://github.com/openHPI/codeocean`
[5] siehe: `https://www.docker.com`

verfolgte Lehransatz basiert auf dem Konzept der testgetriebenen Entwicklung und legt den Fokus darauf, dass Studierende das Programmieren in Python schrittweise erlernen, indem sie eigenständig Aufgaben bearbeiten. Notwendige grundlegende Programmierkonzepte werden durch Kick-off-Seminare und einen Online-Kurs des openHPI[6] vermittelt.

Die Übungsreihe ist in ein praktisches Anwendungsszenario eingebettet und erfordert einen iterativen Entwicklungsansatz. Sie besteht aus fünf Teilaufgaben, die verteilt über das Semester hinweg zu bearbeiten sind. Begleitet werden die Studierenden durch Tutoren. Das übergeordnete Ziel besteht darin, ein Konsolenprogramm zu entwickeln, das wesentliche Funktionen eines Fahrkartenautomaten umsetzt. Zu jeder Teilaufgabe erhalten die Studierenden einen vorgegebenen Entwicklungsstand des Programms und eine textuelle Beschreibung der Anforderungen, die das Programm zusätzlich erfüllen soll. Alle Informationen inkl. des Quellcodes werden über CodeOcean zur Verfügung gestellt. Während der einzelnen Implementierungsphasen haben die Studierenden jederzeit die Möglichkeit, ihre Lösungen zu testen. Für dieses Autograding haben wir aufgabenspezifische Unittests entwickelt, die individuelles Feedback liefern, einschließlich detaillierter Meldungen, wenn Anforderungen nicht oder unzureichend erfüllt werden [Pa22]. Die erwarteten Ausgaben des Konsolenprogramms werden getrennt von den Tests im JSON-Format definiert. Dadurch lassen sich Aufgabenvariationen des Fahrkartenautomaten, die keine funktionalen Änderungen vornehmen, leicht umsetzen. Um sicherzustellen, dass die Studierenden wesentliche Good Practices zur Codeformatierung und -strukturierung einhalten und erlernen, erfolgen mit Hilfe eines Linters zusätzlich aufgabenunabhängige Syntaxtests und Tests des Programmierstils.

Die Python-Übungsreihe haben wir in semesterbegleitenden Übungen in zwei Studierendenjahrgängen erprobt, an denen insgesamt etwa 170 Studierende teilnahmen. Zur Evaluation führten wir Befragungen durch. Aus diesen sowie aus unseren Beobachtungen im Lernprozess wurde deutlich, dass die Verwendung von CodeOcean einen maßgeblichen Beitrag zur Zufriedenheit der Studierenden geleistet hat. Die Möglichkeit, kontinuierlich automatisches Feedback zum Quellcode abzurufen, wurde als äußerst positiv bewertet. Mithilfe des Feedback konnten die Studierenden Fehler selbst erkennen und ihre Lösungen kontinuierlich verbessern. Allerdings gab es auch Kritikpunkte, die sich vor allem auf ungenaues, verwirrendes oder nicht gut nachvollziehbares Feedback sowie unklare Aufgabenstellungen bezogen. Einige Studierende bezeichneten das Feedback als überladen, da es teils viele oder zu detaillierte Fehlermeldungen enthielt. Um diese Probleme anzugehen, fand eine Überarbeitung der Aufgabenstellungen und aller Unittests statt. In den Aufgaben wurden u. a. mehr Vorgaben eingeführt und diese präziser erläutert. Zudem haben wir das individuelle Feedback übersichtlicher und transparenter gestaltet, z. B. durch eine eindeutige Darstellung der In- und Outputs der Testfunktionen und die Ergänzung exemplarischer Lösungshinweise. Dadurch war eine einfachere Fehleranalyse und -behebung für die Studierenden möglich.

Mit dem zweiten Studierendenjahrgang führten wir als zusätzliche Überprüfung des Lernfortschritts nach dem Absolvieren der Übungsreihe einen Abschlusstest durch. Dadurch

[6] https://open.hpi.de/courses/pythonjunior-schule2022

sammelten wir erste Erfahrungen mit dem Einsatz von CodeOcean in einer prüfungsnahen Situation. Die Studierenden mussten dazu unter Prüfungsbedingungen, d. h. in Präsenz und unter Aufsicht von Lehrenden, ihre Programmierfähigkeiten durch das Lösen von zwei zufällig ausgewählten Aufgaben aus einer Aufgabensammlung nachweisen. Dafür wurden weniger komplexe Aufgaben als bei der semesterbegleitenden Übungsreihe verwendet, wie z. B. das Potenzieren zweier Zahlen. CodeOcean erwies sich auch hier als effektives Werkzeug, die Lösungen der Studierenden zu bewerten und ihnen Feedback zu geben. Die anschließenden Rückmeldungen der Studierenden zu dem Einsatz von CodeOcean in dieser prüfungsnahen Situation waren sehr positiv. Daher planen wir, diese Art des Einsatzes weiter auszubauen und in den kommenden Semestern eine digitale Abschlussprüfung mit CodeOcean zu entwickeln und zu erproben.

4 Integration in den Kurs Algorithmen und Programmierung (AuP)

In AuP erwerben Studierende verschiedener Ingenieurstudiengänge Grundlagen der Anwendung ausgewählter Algorithmen und Datenstrukturen sowie der Programmierung mit Java. Dabei arbeiten sie oft selbständig mit Programmier- und Verständnisaufgaben sowie entsprechenden Beispiellösungen. Da Anfängern das kritische Beurteilen eigener Programmierlösungen meist schwer fällt, integrierten wir CodeOcean in den Lehrbetrieb. Nun steht Feedback auch Studierenden zur Verfügung, die nicht an Präsenzübungen/ -tutorien teilnehmen. Zudem werden Betreuende von Kontrollaufgaben entlastet und gewinnen Zeit zur Klärung von Verständnisfragen oder für individuelle Lösungshinweise. Links im LMS sowie den PDF-Aufgabenblättern sollen zum Einreichen der Lösung in CodeOcean anregen. Insgesamt hat etwa die Hälfte der ca. 100 aktiv am Kurs teilnehmenden Studierenden CodeOcean regelmäßig benutzt.

Die Umsetzung der Analyse- und Bewertungsansätze für das Autograding wird durch die von CodeOcean bereitgestellte Infrastruktur sehr erleichtert. Dies ermöglichte uns die Implementierung von Tests, welche auf spezielle Anforderungen von Programmieranfängern ausgerichtet sind. Gleichzeitig gelang es, den Aufwand für die Testerstellung und -pflege gering zu halten. Wichtigste Anforderung an den AuP-Autograder war, dass er auch für Teillösungen (kompilierbar, aber einige geforderte Member fehlen) einsetzbar ist. Dies impliziert, dass herkömmliche JUnit-Tests, welche zur Laufzeit komplette studentische Lösungen benötigen, nicht geeignet sind. Der von uns gewählte Ansatz beruht auf der *reflection*-basierten Analyse studentischer Lösungen, siehe [Dö23]. Wenn Tests aufgrund fehlender Lösungsteile nicht ausgeführt werden können, dann wird dies mitgeteilt. Aus lernmethodischer Sicht ist dies sinnvoll, denn die Studierenden können weiterhin in ihrer gewohnten IDE (z. B. Eclipse) arbeiten und bei Bedarf (Teil-) Lösungen via CodeOcean bewerten lassen. Dabei ist eine ihrem Wissensstand angemessene Formulierung der Fehlermeldungen sehr wichtig. Insbesondere bei Funktionstests sind zur Nachvollziehbarkeit auch die verwendeten Testdaten (Methodenparameter und initialer Zustand von Member-Variablen) sowie die festgestellten und erwarteten Ergebnisse (Ausgaben, Rückgaben, Variablenwerte) mitzuteilen.

Aktuell verwendet der AuP-Autograder eine Java-Einbindung (in CodeOcean „Adapter“) für JUnit. Dazu emuliert er die JUnit-Fehlerausgaben. Positives Feedback und Verbesserungshinweise lassen sich so aber nicht zurückgeben. Deshalb streben wir die Entwicklung eines Adapters an, der die vom Autograder vergebenen Punkte und das textuelle Feedback entgegen nimmt. Zudem sollte der Autograder Formatierungen (insbesondere Hervorhebungen) ins Feedback integrieren können, um so die Lesbarkeit für Studierende zu verbessern.

5 Erprobung automatisierter E2E-Tests in einem WebDev-Kurs

In einem studiengangübergreifenden Grundlagenkurs zur Entwicklung web-basierter Kommunikationsangebote erlernen die Studierenden, Inhalte mit HTML5 strukturiert zu beschreiben, mittels CSS3 zu gestalten sowie mit Hilfe von Javscript zu manipulieren und Nutzereingaben persistent zu halten. Anhand von Übungsaufgaben erarbeitete Kompetenzen sollen sie abschließend in einem eigenen Web-Projekt zusammenführen. Dieser Anwendungsfall erweitert das in [Au21] beschriebene Szenario, indem bereits die strukturierte Beschreibung von Webseiten-Inhalten und auch die (serverseitige) Datenhaltung als Testfall betrachtet wird. Wir adaptierten den dafür vorgeschlagenen Ansatz, den Kompetenzerwerb mittels end-to-end (E2E)-Tests zu überprüfen (vgl. [Le16]), und realisierten ihn in CodeOcean mithilfe einer speziellen Ausführungsumgebung.

Das von CodeOcean empfohlene Container-Basis-Image[7] erweiterten wir zu einer für automatisierte E2E-Tests geeigneten Single-Container-Ausführungsumgebung. Konkret ergänzten wir einen lokalen Webserver zur Bereitstellung der zu prüfenden Lösung sowie einen über das Selenium-Webdriver-Protokoll gesteuerten Webbrowser zur Ausführung der einzelnen Unit-Tests (vgl. [Pa22]). Die zu testende Lösung wird durch CodeOcean als Ordner im Dateisystem der Ausführungsumgebung bereitgestellt. Den lokalen Webserver konfigurierten wir deshalb so, dass er diesen Ordner als Basisverzeichnis verwendet.

Für den produktiven Einsatz im nächsten Lehrzyklus mit ca. 150 Studierenden übertragen wir folgende Beispielfälle in CodeOcean. In Use Case 1 soll ein HTML-Grundgerüst gemäß eines in der Aufgabenstellung beschriebenen Wireframe-Modells um typische Inhaltsbereiche mittels HTML5-Container-Elementen ergänzt werden. Der Funktionstest traversiert die DOM-Struktur und gleicht sie mit einem Musterdatensatz im JSON-Format ab. Das Feedback adressiert fehlende oder überzählige Bereiche sowie ungünstig gewählte Container-Elemente. In Use Case 2 sollen komplex strukturierte Inhaltsformen, wie Wort-Bild-Kombinationen, Tabellen oder Aufzählungen, in eine elementar strukturierte Webseite eingefügt werden. Der Funktionstest vergleicht die DOM-Teilstruktur mit einem Musterdatensatz im JSON-Format. Das Feedback adressiert Fehler in der Wahl und Verschachtelung geeigneter HTML-Elemente. In Use Case 3 soll mittels *unobtrusive Javascript* (vgl. [Ko14]) bestimmten, mit DOM-Methoden zu selektierenden Elementen einer vorgegebenen HTML-Struktur ein Eventlistener hinzugefügt werden, der eine vorgegebene Eventhandler-Funktion

[7] siehe: `https://github.com/openHPI/dockerfiles`

für eine spezifische Interaktion registriert. Der Funktionstest simuliert die vorgegebene Nutzerinteraktion und prüft das Eintreten der vom vorgegebenen Eventhandler vorgenommenen Inhaltsänderung. Das Feedback adressiert Fehler in der Selektion der korrekten Elemente sowie der korrekten Registrierung des Eventlisteners. In Use Case 4 soll für eine vorgegebene Webseite mit bereits hinzugefügten Eventlistenern der Funktionsrumpf des registrierten Eventhandlers in Javascript ergänzt werden. Der Funktionstest vergleicht die nach erfolgreicher Ausführung des Eventhandlers eingetretene Inhaltsänderung mit der laut Aufgabenstellung erwarteten Änderung. Das Feedback adressiert Fehler im Programmfluss, soweit diese aus dem Vergleich von tatsächlicher mit erwarteter Ereignisantwort abgeleitet werden können.

Probleme bereitete uns zunächst der Einsatz der Ausführungsumgebung als Single-Container-Lösung mit mehreren gleichzeitig aktiven Anwendungen. Das CodeOcean-Entwicklerteam löste dieses Problem jedoch mittels Funktionsupdate[8]. Somit ist die Umsetzung von Aufgaben zur testgetriebenen Web-Entwicklung mittels automatisierter E2E-Tests in CodeOcean möglich. Mit Hilfe von parametrisierten Tests konnten wir das Feedback zu Lösungseinreichungen deutlich differenzierter gestalten.

6 Systemnahe Programmierung in Rust

Bei der systemnahen Softwareentwicklung kommen oft Programmiersprachen zum Einsatz, die Studierende weder aus dem Grundlagenstudium noch aus ihrer persönlichen Erfahrung beherrschen. Eine solche Programmiersprache ist Rust [MK14]. Sie zeichnet sich insbesondere durch eine Forcierung von robustem und effizientem Code aus. Damit empfiehlt sich Rust etwa zur Implementierung von Betriebssystemmechanismen im Kurs „Advanced Operating Systems“, trotz einer für Einsteiger vergleichsweise steilen Lernkurve [Ru18]. Im Kursverlauf bearbeiten die Studierenden in teamorientierten Workshops konzeptionelle und algorithmische Probleme, deren Lösung dann auf individuelle Programmiertätigkeiten ohne unmittelbares Lehrendenfeedback abgebildet wird. Diese Dualität der Lernziele – sowohl im domänenspezifischen Wissen und dessen Anwendung, als auch in der hierfür erforderlichen Programmiersprache – stellt uns Lehrende jedoch vor ein Ressourcenproblem. Existierendes Online-Material,[9] welches auf das individuelle Erlernen von Rust anhand generischer Anwendersoftware ausgelegt ist, kann dieses Problem nicht unter Wahrung des didaktischen Leitgedankens kooperativen, problemorientierten Lernens lösen. Zugleich soll die Entwicklung und Pflege einer spezialisierten Insellösung vermieden werden.

Um Rust-Programmierung motivierend, effizient und stoffgerecht zu lehren, lag ein Ziel dieses Anwendungsfalls daher in der Erweiterung von CodeOcean um diese bislang nicht unterstützte Programmiersprache und wurde von uns, mit Unterstützung durch das HPI, in zwei Schritten umgesetzt: (1) Erstellung eines zugeschnittenen Container-Images als

[8] vgl.: https://github.com/openHPI/poseidon/blob/main/docs/configuration.md#supported-docker-images
[9] Bspw. https://github.com/rust-lang/rustlings, https://doc.rust-lang.org/stable/rust-by-example/.

Grundlage einer neuen Ausführungsumgebung. Dieses stellt alle Sprachtools sowie, für systemnahe Entwicklung besonders relevant, Bibliotheken und Hardwareabstraktionen entsprechend der Aufgabenstellung bereit. (2) Implementierung eines CodeOcean-Adapters für das von Rust genutzte Build-Tool *cargo* in Ruby. Hierfür müssen die Gesamtzahl ausgeführter Tests, die Anzahl fehlgeschlagener Tests sowie etwaige Fehlermeldungen aus dem Container an CodeOcean übergeben werden. Bislang wurde dies mithilfe regulärer Ausdrücke implementiert, welche menschenlesbare Ausgaben eines Build-Tools filtern. Da jedoch *cargo* als Build-Tool JSON zur strukturierten Datenrepräsentation unterstützt, lassen wir stattdessen diese Ausgabe mittels eines Ruby-Standardmoduls im Adapter parsen. Somit können für zukünftige Erweiterungen des CodeOcean-Frontends auch weitere Informationen bereitgestellt werden, beispielsweise Hilfestellungen für typische Programmierfehler, welche durch *cargo* bei nicht kompilierbaren Programmen ausgegeben werden.

Allerdings mussten Automatisierungsfunktionen der Rust-Tools stellenweise limitiert werden: Da *cargo* darauf ausgelegt ist, Codeabhängigkeiten durch direkten Download der benötigten externen Quellen (Rust „Crates“) aufzulösen, muss sowohl die Ausführungsumgebung für Netzwerkzugriff konfiguriert sein, als auch der Host entsprechende Berechtigungen gewähren. Ersteres erwies sich in Machbarkeitstests mit verschiedenen Container-Images als problematisch, während Letzteres aus Sicherheitsgründen fragwürdig ist. Es hat sich daher für unseren Kurs als sinnvoll erwiesen, mehrere Ausführungsumgebungen mit verschiedenen Container-Images zu verwenden: je nachdem, ob bspw. Interprozesskommunikation, Dateisystemzugriffe oder (simulierte) Netzwerkkommunikation im Fokus der jeweiligen Aufgabe stehen, setzen wir hierfür angepasste und mit Offline-Versionen der jeweils benötigten Crates bestückte Images ein. Dies bietet den didaktischen Mehrwert, die Studierenden durch die Beschränkung auf konkret vorgegebene Drittsoftware an realistische Arbeitsbedingungen in der Praxis zu gewöhnen. Darüber hinaus können wir das Spektrum möglicher Lösungsvarianten durch verbindliche Vorgabe bestimmter APIs sinnvoll eingrenzen, was wiederum die Entwicklung passgenauer Tests begünstigt.

Im Rahmen des nächsten Lehrzyklus ist geplant, eine Rust-Übungsreihe in CodeOcean mit ca. 20 Teilnehmern prototypisch umzusetzen. Wir erwarten hieraus weitergehendes Feedback aus Teilnehmersicht, welches bspw. Impulse zur ergonomischen Neugestaltung des Frontends geben kann, um Rust-spezifische Werkzeugunterstützung in den Ausgaben umfassender zu berücksichtigen.

7 Zusammenfassung und Ausblick

Die Erprobung von CodeOcean in mehreren Kursen mit jeweils spezifischen Anforderungen unter realen Bedingungen hat gezeigt, dass bis dahin separat entwickelte Ansätze in einem einheitlichen System abgebildet werden können. Dies ermöglicht eine homogene, hochschulweite Lösung durch die viele einzelne Insellösungen vermieden werden können. Infolgedessen kann administrativer Aufwand zentralisiert und Einarbeitungsaufwand von Studierenden reduziert werden. Die Machbarkeit konnten wir in vier Anwendungsszenarien

auf dem Technology Readiness Level TRL7[10] prüfen und qualitativ bestätigen. Es zeigte sich, dass neue, vom Entwicklungsteam des HPI nicht vorhergesehene Anwendungsfälle mit vertretbarem Aufwand in CodeOcean abgebildet werden können. Entwicklungsbedarf besteht mit Blick auf die Verbesserung der Nutzungserfahrung für die Lernenden. Der Einsatz gängiger Test-Werkzeuge kommt bei der Erzeugung hilfreichen Feedbacks zum richtigen Zeitpunkt (vgl. [Je22]) für getestete Lösungseinreichungen an seine Grenzen. Unsere weiterführenden Untersuchungen befassen sich daher aufbauend auf Ergebnissen der quantitativen Beurteilung im Rahmen der kontinuierlichen Lehrevaluation u. a. mit der Weiterentwicklung der Nutzer-Schnittstelle von CodeOcean sowie der Generierung differenzierten Feedbacks.

Literaturverzeichnis

[Au21] Aubele, L.; Martin, L.; Hirmer, T.; Henrich, A.: An Architecture for the Automated Assessment of Web Programming Tasks. In. 5. ABP-Workshop. 2021.

[Dö23] Döring, U.: JUnit free checks of Java solutions in CodeOcean, https://github.com/udoering/JavaSolCheckCOJU, 2023.

[Je22] Jeuring, J. et al.: Towards Giving Timely Formative Feedback and Hints to Novice Programmers. In: ITiCSE-WGR '22. S. 95–115, 2022.

[Ko14] Koch, P.-P.: The principles of unobtrusive JavaScript, W3C Wiki, 9. Okt. 2014, URL: https://t.ly/Ym0lh.

[Le16] Leotta, M. et al.: Approaches and Tools for Automated End-to-End Web Testing. In: Advances in Computers. Bd. 101, Elsevier, S. 193–237, 1. Jan. 2016.

[MK14] Matsakis, N. D.; Klock, F. S.: The Rust Language. In: Proc. 2014 ACM SIGAda Ann. Conf. on High Integrity Language Technology. HILT '14, S. 103–104, 2014.

[Pa22] Pajankar, A.: Python Unit Test Automation: Automate, Organize, and Execute Unit Tests in Python. Apress, Berkeley, CA, 2022.

[RRH12] Rodríguez-del-Pino, J. C.; Rubio Royo, E.; Hernández Figueroa, Z.: A Virtual Programming Lab for Moodle with automatic assessment and anti-plagiarism features. In: Proc. EEE 2012. 2012.

[Ru18] Rust Foundation: Rust survey 2018 results, Nov. 2018, URL: https://blog.rust-lang.org/2018/11/27/Rust-survey-2018.html, Stand: 12. 06. 2023.

[Se21] Serth, S. et al.: Improving the Scalability and Security of Execution Environments for Auto-Graders in the Context of MOOCs. In. 5. ABP-Workshop. 2021.

[SS22] Strickroth, S.; Striewe, M.: Building a Corpus of Task-Based Grading and Feedback Systems for Learning and Teaching Programming. iJEP 12/5, S. 26–41, 2022.

[10] vgl. https://op.europa.eu/s/yNrL, Kap. 3

Automatisiertes Bewerten bei der praktischen Vermittlung von Methoden des Maschinellen Lernens

Katharina Holstein[1], Nata Kozaeva[2] und Korinna Bade[3]

Abstract: Im Kontext der Informatiklehre für die Einführung in das Maschinelle Lernen an Hochschulen für angewandte Wissenschaften liegt ein wesentlicher Fokus auf Anwendungsszenarien und Datensätzen mit aktuellem Problembezug. Zeitgleich ist der Stand der Vorkenntnisse vor allem der Masterstudierenden sehr divers. Dazu wird im folgenden Paper dargestellt, wie die praktischen Vermittlung von Methoden des Maschinellen Lernens mit Python durch automatisiertes Testen unterstützt werden kann. Dabei liegt ein besonderer Fokus auf Lösungen für die speziellen Anforderungen der Programmierausbildung für das Maschinelle Lernen.

Keywords: Selbstlernprogrammierkurs; Maschinelles Lernen; Automatisiertes Testen; VPL

1 Einleitung

An der Hochschule Anhalt werden in verschiedenen Studiengängen und Modulen Kenntnisse zu Themen des Maschinellen Lernens und des Data Science vermittelt. Für die im Folgenden präsentierte Arbeit ist insbesondere der Masterstudiengang Data Science von Interesse, in welchen Studierenden beliebiger Bachelor-Abschlüsse zugelassen werden. Daraus ergibt sich eine sehr heterogene Studierendenschaft mit unterschiedlich vorhandenen Kompetenzen im Bereich der Programmierung. Von Studienanfängern ohne jegliche Programmiererfahrung bis hin zu Studierenden mit einem Informatikabschluss ist alles vertreten. Entsprechend didaktisch anspruchsvoll ist der Umgang damit. Grundsätzlich wird in einem speziell dafür konzipiertem ersten Semester Studienanfängern mit Bachelor-Abschlüssen jenseits der Informatik ein Lehrangebot unterbreitet, welches darauf abzielt, die wichtigsten Grundkompetenzen der Informatik und so auch der Programmierung zu vermitteln. Nichtsdestotrotz muss innerhalb der Module der unterschiedlichen Lernkurve der einzelnen Studierenden Rechnung getragen werden. Studierende brauchen auf Grund ihres unterschiedlichen Wissensstandes meist sehr individuelle Unterstützung und gleichzeitig viel Programmierpraxis. Da dies mit den vorhandenen personellen Ressourcen sehr schwierig zu leisten ist, soll automatisiertes Bewerten von Programmieraufgaben ein zeitnahes Feedback und damit Unterstützung bei der Erlangung von Programmierpraxis geben.

In der vorliegenden Arbeit wird insbesondere die Vermittlung von Programmierkenntnissen im Umfeld des Maschinellen Lernens und des Data Science betrachtet. Hieraus ergeben

[1] Hochschule Anhalt, Fachbereich Informatik und Sprachen, Lohmannstraße 23, 06366 Köthen, Deutschland; Katharina.Holstein@hs-anhalt.de
[2] Hochschule Anhalt, Fachbereich Informatik und Sprachen, Lohmannstraße 23, 06366 Köthen, Deutschland
[3] Hochschule Anhalt, Fachbereich Informatik und Sprachen, Lohmannstraße 23, 06366 Köthen, Deutschland; Korinna.Bade@hs-anhalt.de; https://orcid.org/0000-0001-9139-8947

doi:10.18420/abp2023-10

sich spezifische Anforderungen an das automatische Testen, welche im Folgenden näher betrachtet werden sollen. Die genutzte Programmiersprache ist Python, die in dieser Domäne hohe Relevanz hat. Im Bezug zu anderen Arbeiten ist festzuhalten, dass es zwar verschiedene Systeme zum automatisierten Testen von Python-Code gibt, die speziellen Anforderungen im Bereich des Maschinellen Lernens und des Data Science bisher unseres Wissens nach aber in der Literatur noch nicht explizit betrachtet wurden. Im Folgenden werden zunächst unsere Lernszenarien genauer vorgestellt. Anschließend werden die bisher von uns identifizierten Herausforderungen an das automatisierte Testen im Data Science Umfeld sowie unser Lösungsansatz an einzelnen Beispielen dargestellt.

2 Lernszenarien und Stand der Arbeit

Es werden zwei Lernszenarien betrachtet, die unterschiedlich stark das automatisierte Testen als Bestandteil integrieren. Dabei dient das erste Szenario dazu, Studierende an das Arbeiten mit automatisierten Tests heranzuführen und eine „herkömmliche“ Lehrveranstaltung anzureichern sowie in der Interaktion mit den Studierenden Erfahrungen im Umgang mit automatisierten Tests zu sammeln und weitere Anforderungen abzuleiten. Im zweiten Lernszenario soll dies entsprechend weitergeführt werden, so dass hier dem automatisierten Testen eine zentrale Rolle zukommt. Dabei soll das automatisierte Testen nicht zum Zwecke der Notenfindung eingesetzt werden. Stattdessen dient es der Unterstützung des Selbststudiums und soll mit möglichst sinnvollen und konkreten Fehlermeldungen direktes, zeitnahes Feedback an die Studierenden weitergeben.

Lernszenario 1: Unterstützung der Praktika des Moduls Maschinelles Lernen

Die Lehrveranstaltung Maschinelles Lernen wird bereits im ersten Mastersemester, zeitgleich mit einem Modul zur Vermittlung von Programmierfähigkeiten, durchgeführt. Programmierfertigkeiten der Studierenden werden auch in diesem Modul mit entwickelt, weiter vertieft und ausgebaut. Ein umfangreiches Verständnis von entsprechenden Bibliotheken aus dem Bereich Data Science kann nicht vorausgesetzt werden. Inhaltlich gibt die Vorlesung eine Einführung in die Grundlagen des Maschinellen Lernens und stellt einzelne Ansätze des überwachten und unüberwachten Lernens sowie deren praktische Umsetzung mit Python vor. Bestandteil der Lehrveranstaltung ist ein Programmierpraktikum, welches auf verschiedenen Anforderungsniveaus unterschiedliche Programmieraufgaben vorsieht. Um die Präsenzzeiten der Praktika zielgerichteter zu gestalten und den Studierenden auch bei der Arbeit zu Hause eine Unterstützung zu geben, sollen die Lösungen zu den Praktikumsaufgaben mit automatisierten Tests geprüft werden. Das Feedback soll typische Fehler der Studierenden zeitnah identifizieren, damit die Studierenden ausreichend Zeit für die eigentliche Lösung des Problems einsetzen können.

Lernszenario 2: Selbstlernkurs zur Vermittlung von Grundlagen der Python-Programmierung im Umfeld von Data Science

Das zweite Lernszenario ist als Ergänzungsmaterial konzipiert und soll daher unabhängig für sich ohne Betreuung durch eine Lehrperson absolvierbar sein. In diesem Kurs soll insbesondere der Umgang mit speziellen Bibliotheken aus dem Umfeld des Data Science, wie z. B. numpy, pandas, matplotlib oder ähnliches trainiert werden. Grundsätzlich lassen sich diese Inhalte auch über Tutorials im Web erarbeiteten. Dies stellt jedoch vor allem Studierende ohne einen Informatikhintergrund vor große Herausforderungen, da sie häufig auf Grund der Fülle der verfügbaren Informationen und fehlendem Wissen Tutorials nicht gezielt recherchieren, Dokumentationen nicht nachvollziehen und ähnliche Problemstellungen nicht übertragen können sowie wesentliche Bibliotheken nicht betrachten. Daher soll das vorliegende Lernszenario die Studierenden beim Selbststudium unterstützen und idealerweise einen Übergang in das selbstgesteuerte Lernen mit Internetressourcen bilden.

Aktueller Stand der Umsetzung

Als erster Schritt erfolgte bisher die Umsetzung des ersten Lernszenarios. Dieses wird im kommenden Semester mit den Studierenden praktisch erprobt, evaluiert und wo notwendig, verbessert und erweitert. Daran anschließend wird das Lernszenario 2 umgesetzt. Alle Materialien sollen als Open Educational Ressource (OER) umgesetzt werden und stehen damit anderen Lehrenden (und natürlich auch Lernenden) zur Verfügung.

3 Automatisches Testen für Szenarien des Maschinellen Lernens

Für die Umsetzung kann prinzipiell ein beliebiges Tool zum Bewerten von Python-Code zum Einsatz kommen. Die besonderen Herausforderungen müssen durch die Umsetzung der eigentlichen Tests gelöst werden. Da an der Hochschule Anhalt Moodle als Lernmanagementsystem mit dem Plugin „Virtual Programming Lab (VPL)“ [Ro23] bereits in einer anderen Lehrveranstaltung für Tests von Java-Programmen im Einsatz ist, wurde diese Umgebung für die Umsetzung gewählt. Das Plugin stellt im Browser einen Editor bereit, in welchem die Studierenden Ihren Quellcode in Form von Python-Skripten eingeben und die Tests über die Nutzerschnittstelle durchführen können. Das VPL-Plugin nutzt im Hintergrund den Zugriff auf einen Jail-Server, welcher die Tests der eingereichten Python-Skripte mittels Standardunittests [Ro01]. übernimmt. Dazu werden auf dem Server die jeweiligen Test-Skripte sowie weitere benötigte Ressourcen wie etwa Standard-Datensätze hinterlegt.

Im Lernszenario 1 erfolgt der Einstieg in das automatisierte Testen begleitet. Im ersten Praktikum erfolgt zunächst eine *Einführung in die grundsätzlich verwendeten Tools*. Für den Einstieg in das automatisierte Testen wird der Zugriff auf den Jailserver sowie die Verwendung des VPL-Plugins in Moodle an einem einfachen „Hello World“ Testskript erläutert. Der Fokus liegt zunächst auf der Einbindung der eigenen Skripte, deren Evaluierung, dem Verständnis der Rückgabewerte bzw. Fehlermeldungen sowie dem grundsätzlichen Verständnis zur Handhabung des automatisierten Testens. Darüber hinaus wird die Integration eigener Datensätze besprochen, welches ebenfalls über das Plugin möglich ist.

Erste einfache Aufgaben werden im Rahmen des Praktikums selbst gelöst, um die Einstiegshürde zu senken, da bei Problemen direkt die Lehrkraft zur Verfügung steht. Weiterführende Programmieraufgaben müssen zum nächsten Praktikum im Selbststudium vorbereitet und dort präsentiert werden. Dabei sollen die automatisierten Tests unterstützen. In den folgenden Praktika präsentieren verschiedene Studierende ihre Lösungen zur Anregung einer Diskussion über Lösungsstrategien und somit der Schaffung des Rahmens eines Flipped Classroom [Sp16]. Der Fokus liegt dabei nicht auf dem Finden des besten Lösungsansatzes, sondern in der Bildung von Verständnis für verschiedene Ansätze, deren Vor- und Nachteile sowie der Schaffung einer Lernumgebung, in der unterschiedliche Herangehensweisen gleich honoriert werden. Darüber hinaus soll diese Herangehensweise verhindern, dass Studierende Lösungsstrategien lediglich konsumieren oder kopieren und stattdessen auch eigene Strategien präsentieren.

Alle nachfolgenden Praktikumseinheiten folgen dem gleichen **Grundaufbau**, um den Lernprozess der Studierenden zu unterstützen. Basis bildet das jeweilige Vorlesungsskript zum aktuellen Thema, welches auch jeweils ein praktisches Beispiel enthält. Auf dessen Basis soll zunächst ein Skript mit einem anderen, durch die Studierenden frei gewählten Datensatz mit gleicher Funktionalität umgesetzt werden (Schritt I). Im nächsten Schritt (Schritt II) sollen kleinere Veränderungen am Code vorgenommen werden, um ein Verständnis für die genutzten Bibliotheken und Tools, Auswirkungen von Parametern etc. aufzubauen. Hierfür sollen die Studierenden sowohl auf die Dokumentation der Bibliothek zurückgreifen als auch entsprechende Standardforen bzw. Hilfsseiten nutzen. Im letzten Schritt (Schritt III) sollen dann einzelne Algorithmen selbst implementiert werden, ohne auf vorgefertigte Bibliotheken zurückzugreifen, beginnend bei einzelnen Funktionen wie z. B. die Berechnung der Entropie eines Knotens in Entscheidungsbäumen bis hin zur vollständigen Umsetzung eines Algorithmus wie etwa der gesamte ID3-Algorithmus.

Aus diesem Grundaufbau ergeben sich spezielle Anforderungen an die konkrete Ausgestaltung des automatischen Bewertens, welche im Folgenden näher betrachtet und mit Umsetzungsbeispielen untersetzt werden sollen. Eine große Rolle im Bereich des Maschinellen Lernens spielt die **Arbeit mit Datensätzen**. Dabei sollen die Studierenden mit möglichst unterschiedlichen Datensätze in Berührung kommen. Im Praktikum wird dies durch die beliebige Wahl von Datensätzen ermöglicht. Neben Standarddatensätze, wie z. B. aus dem UCI-Repository [KLN23], sollen auch eigene Datensätze bzw. Datensätze aus der Industrie, von wissenschaftlichen Partnereinrichtungen oder aus den Fachbereichen der Hochschule verwendet werden können. Dies stellt für das automatisierte Bewerten eine besondere Herausforderung dar, da die Korrektheit der Lösung unabhängig vom spezifischen Datensatz geprüft werden soll. Notwendige Schritte, z. B. in der Datenvorverarbeitung, sind aber ggf. stark vom Datensatz, der Ausprägung der Feature u. ä. abhängig. Die Testumgebung muss möglichst generisch die notwendigen Schritte im Zusammenhang mit dem jeweils genutzten Datensatz bewerten können. Durch die Verwendung unterschiedlicher Datensätze können verschiedene Schwierigkeitsgrade für die Studierenden realisiert und unterschiedlichen Vorkenntnissen, aber auch Interessen Rechnung getragen werden. Die Verwendung generischer

automatisierter Tests reduziert zusätzlich den Aufwand des Lehrpersonals in der Betreuung, da der Datensatz selbst nicht in der kompletten Tiefe durchdrungen oder vorbereitet werden muss.

Im Folgenden sollen zwei Herausforderungen im Umgang mit generischen Datensätzen sowie die Umsetzung der Lösung gezeigt werden. Für Klassifikationsaufgaben ist die *Identifizierung der Klassenspalte* in den Daten erforderlich. Grundsätzlich könnte man natürlich verlangen, dass Studierende ihre Daten vor der Nutzung in ein bestimmtes Format überführen müssen, aus dem das Zielattribut direkt hervorgeht, bzw. das Zielattribut direkt benennen müssen. Dies erfordert jedoch ggf. bereits eine gewisse Expertise im Umgang mit Datensätzen. Um die Einstiegshürde zu senken und Fehler bei der händischen Behandlung der Daten zu vermeiden bzw. erkennen zu können, sollen die Test-Skripte eine automatisierte Erkennung des Zielattributs umsetzen. So kann auch überprüft werden, ob die Studierenden überhaupt das richtige Zielattribut identifiziert haben und daraus abgeleitet Zielattribute und Merkmalsklassen korrekt erzeugt und kodiert haben sowie Trainings- und Testdaten korrekt abgeleitet wurden. Dazu wurde basierend auf den uns vorliegenden Datensätzen eine Heuristik abgeleitet, die typische Ausprägungen von Datensätzen berücksichtigt. Die entsprechende Funktion, die in den Test-Skripten zum Einsatz kommt, ist in List 1 im Pseudocode dargestellt. Im Falle des Fehlschlags der Klassenerkennung wird eine Fehlermeldung ausgegeben, die eine entsprechende Merkmalsbenennung durch den Studierenden erfordert. Um Fehlern vorzubeugen, überprüft das Skript aber auch die formelle Eignung der benannten Klasse. Gleichzeitig werden diese Datensätze dann betrachtet, um die Heuristik weiter zu verbessern.

Ein weiterer wichtiger Schritt der Datenvorverarbeitung ist das *Feature Encoding*. Nicht numerische Feature müssen für viele Algorithmen zunächst in numerische Feature umgewandelt werden. Auf der anderen Seite dürfen bereits numerische Feature nicht noch einmal encodiert werden. Dies ist zugleich eine typische Fehlerquelle bei Studierenden beim Übertragen von Skripten auf neue Datensätze. List 2 enthält den Pseudocode des eingesetzten Tests. Dabei wird das korrekte Encoding durchgeführt und dann mit der Studierendenlösung verglichen.

Insgesamt wird in Schritt I aller Praktikumsaufgaben ein automatisiertes Testen einzelner Arbeitsschritte der gesamten **Datenvorverarbeitungskette** mittels Unittests vorgenommen. Diese Überprüfung folgt dem Schema: Laden von benötigten Bibliotheken, Verwendung der korrekten Funktion zur Bearbeitung von Datensätzen, Aufspaltung des Datensatzes in Zielattribute und Feature (vgl. List 1), korrekte Verwendung eines Encodings (vgl. List 2), Aufteilung des Datensatzes in Trainings- und Testdaten. Neben den gezeigten Pseudocodes kommen in dieser Verarbeitungskette z. B. die explizite Überprüfung von Datentypen bzw. Python-Klassen oder die Überprüfung der Shapes der Teildatensätze zum Einsatz.

Eine weitere Herausforderung beim automatisierten Testen [Bo17; Fa15; Ly12] liegt in der **Überprüfung unspezifischer Aufgabenstellungen**, wie der Änderung von Visualisierungen zum Explorieren der Bibliotheksfunktionen in Schritt II. Dies kann über die Bibliothek

```
def data_preprocessing_X_y(data: pd.DataFrame) -> List:
  Wenn Datensatz data weniger als 2 Spalten enthält:
    Gib entsprechende Fehlermeldung zurück
  Wenn Datensatz data mind. eine Spalte 'class', 'klasse' oder 'target'
      enthält:
    Wenn es genau eine solche Spalte gibt:
      Wenn diese Spalte die Eigenschaften einer Klassenspalte erfüllt (nur
          diskrete Werte):
        Verwende diese Spalte als Zielattribut y und den Rest als Daten X
            und gib [X,y] zurück
      Sonst: Gib Fehlermeldung zur Spaltenauswahl zurück
    Sonst: Gib Fehlermeldung der Uneindeutigkeit zurück
  Wenn Datensatz data mind. eine einzelne Spalte vom Datentyp Object
      enthält:
    Wenn es genau eine solche Spalte gibt:
      Verwende diese Spalte als Zielattribut y und den Rest als Daten X und
          gib [X,y] zurück
    Sonst: Gib Fehlermeldung der Uneindeutigkeit zurück
  Wenn Datensatz data mindestens eine Spalte enthaelt, die nur
      Integer-Werte besitzt:
    Verwende von allen Spalten, die diese Bedingung erfüllen, diejenige,
        die die wenigsten unterschiedlichen, eindeutigen Werte hat, als
        Zielattribut y und den Rest als Daten X und gib [X,y] zurück
  Gib Fehlermeldung zur Klassenidentifizierung zurück
```

List. 1: Generische Bestimmung des Zielattributs in einem Datensatz.

Mock bzw. MagicMock der Unittest Bibliothek gelöst werden. Diese erlaubt es, über eine zentrale Mock-Klasse die Verwendung von Methoden bzw. Attributen und ihrer spezifischen Funktionsaufrufe zu testen, ohne spezifische Abfragen vorzunehmen. Falls eine Änderung der Methoden nicht geschehen ist, werden über entsprechende Fehlermeldungen die Studierenden mit aussagekräftigen Hinweisen auf die korrekte Lösung hingewiesen werden. Der Vorteil liegt hier in der generischen Anwendbarkeit der Testumgebung auf herausgelöste Problemstellungen sowie auf unspezifischen Aufgabenstellungen. In einigen Fällen können Funktionsaufrufe auch über regex Ausdrücke getestet werden. Dies erfordert keine extra Funktion im Studierendenskript, die gemockt werden muss. Allerdings ist diese Methode weniger flexibel, da ein String-Vergleich nur in wenigen Fällen möglich ist.

Als letztes soll die **Bewertung kleinerer Teilfunktionen** im Schritt III für Ergebnisse auf unbekannten Datensätzen betrachtet werden. Dies kann in der Regel darüber evaluiert werden, ob das in der Lösung berechnete Ergebnis in etwa mit dem korrekten Ergebnis übereinstimmt, was durch die „assertAlmostEqual“ Funktion ermöglicht wird. Können die Studierenden mit beliebigen Datensätzen arbeiten, kann hierfür aber nicht ein festes Ergebnis hinterlegt werden. Stattdessen muss, wie oben bereits beschrieben, eine generische Funktion hinterlegt werden, die für jeden möglichen Datensatz das korrekte Ergebnis bestimmen kann. Zeitgleich muss überprüft werden, dass Studierende nur zulässige Bibliotheken

```
def umkodierung_X(attribute_X, data):
  Für jedes Feature in attribute_X:
    Wenn das Feature vom Datentyp Object ist:
      Encodiere die Daten dieser Spalte mit dem LabelEncoder
    Wenn das Feature vom Datentyp Integer oder Float ist:
      Verändere die Daten nicht
    Gib die Umkodierten Daten zurück

def test_encoding_X(self):
  studierende_attribute = die kodierten Daten der Studierendenlösung
  korrekte_umkodierung_X = korrektes Encoding über umkodierung_X
  Für jedes Feature in studierende_attribute:
    Wenn das Feature vom Datentyp Object ist:
      Gib Fehlermeldung zu fehlender Kodierung zurück
  Für jedes Feature in korrekte_umkodierung_X:
    Wenn der Datentyp des Features in studierende_attribute und
        korrekte_umkodierung_X unterschiedlich ist:
      Gib Fehlermeldung zu fehlerhaft durchgeführter Kodierung zurück
    Wenn der Datentyp des Features in studierende_attribute und
        korrekte_umkodierung_X jeweils Integer, die Werte aber
        unterschiedlich sind:
      Gib Fehlermeldung zu fehlerhaft durchgeführter Kodierung zurück
```

List. 2: Überprüfung auf richtige Verwendung des Feature Encodings.

und Funktionen nutzen und nicht etwa eine bereits fertige Funktion eine Bibliothek. Hier helfen sogenannte Search Patterns, mit denen der Einsatz bestimmter Methoden getestet werden kann. Anspruchsvoller wird dieser Abgleich, wenn statt Teilfunktionen *ganze Algorithmen* wie etwa der ID3-Algorithmus umgesetzt werden. Hier muss der Abgleich auf dem gesamten Klassifikationsergebnis erfolgen. Als Zwischenschritt kann der Vergleich von Evaluationsmaßen (wie z. B. Accuracy, Precision, Recall) von Musterlösung und Studierendenlösung einen Hinweis geben, ob sich die Lösung zumindest in der erwarteten Spanne bezogen auf die Qualitätsmaße befindet.

4 Fazit und Ausblick

Für das hier vorgestellte und bereits umgesetzte Lernszenario 1 erfolgt im kommenden Semester eine Evaluierung im Praxistest im Rahmen der Lehrveranstaltung. Unter Einbezug der dort gewonnenen Erkenntnisse, wird nicht nur dieses Szenario weiter verbessert, sondern parallel dazu auch Lernszenario 2 umgesetzt. In diesem werden die Ansätze aus Lernszenario 1 erweitert, um sowohl Grundlagenwissen zum Programmieren in Python mit dem Fokus auf Data Science und Maschinelles Lernen, Datensatz- und Programmierbibliotheksspezifische Inhalte, wie z. B. Visualisierung von Lernergebnissen auf Hyperspektralbildern, Anwendung unterschiedlicher Methoden für unterschiedliche Datensatztypen (Zeitreihen, visuelle oder akustische Datensätze, . . .), als auch die korrekte Auswertung und Darstellung der

Lernergebnisse im jeweiligen datensatzspezifischen Kontext zu vermitteln. Zusätzlich soll die Bearbeitung unscharfer, unbekannter oder heterogener Daten sowie deren Verarbeitung vermittelt werden. Die Verwendung entsprechend realer, größerer Datensätze wird darüber hinaus die Verwendung von rechenstarken Hardwaretools vermitteln, deren Effekte sich bei der Verwendung kleiner Standarddatensätzen nur schwer darstellen lassen.

Die hier vorgestellten Lerninhalte sind als Open Educational Resource konzipiert und werden nach der Evaluierung anderen Lehrenden zur Verfügung gestellt. Darüber hinaus ist eine Datensatzbibliothek aus Datensätzen der angewandten Forschung als auch der Industrie für die Studierenden geplant, welche u. a. für den Selbstlernkurs zur Verfügung stehen wird. Diese Daten decken einen großen Bereich an möglichen mehrdimensionalen und heterogenen oder unvollständigen Daten ab, sodass nach dem Erlangen eines Grundverständnisses zu Datensätzen eine tiefgreifendere Problembehandlung möglich wird. Dies dient auch zur Vorbereitung für die spätere Projektarbeit im Studium, da Studierenden bereits frühzeitig an praxisrelevante Arbeit herangeführt werden.

Literaturverzeichnis

[Bo17] Bott, O. J.; Fricke, P.; Priss, U.; Striewe, M.: Automatisierte Bewertung in der Programmierausbildung. Waxmann, 2017, ISBN: 9783830936060.

[Fa15] Fangohr, H.; O'Brien, N. S.; Prabhakar, A.; Kashyap, A.: Teaching Python programming with automatic assessment and feedback provision. ArXiv abs/1509.03556/, 2015.

[KLN23] Kelly, M.; Longjohn, R.; Nottingham, K.: The UCI Machine Learning Repository, last accessed: 2023-09-07, 2023, URL: https://archive.ics.uci.edu.

[Ly12] Lynch, C.; Ashley, K. D.; Aleven, V.; Pinkwart, N.: Ill-Defined Domains and Adaptive Tutoring Technologies. In: Adaptive Technologies for Training and Education. Cambridge University Press, 2012.

[Ro01] van Rossum, G.: unittest – Unit testing framework, last accessed: 2023-06-08, 2001, URL: https://docs.python.org/3/library/unittest.html.

[Ro23] Rodríguez-del-Pino, J. C.: VPL, the Virtual Programming lab for Moodle, last accessed: 2023-09-07, 2023, URL: https://vpl.dis.ulpgc.es/.

[Sp16] Spannagel, C.: Flipped Classroom, last accessed: 2023-06-07, 2016, URL: https://cspannagel.wordpress.com/category/flippedclassroom-2/.

Deploy-to-Grading: Automatische Bewertung von Programmieraufgaben mit CI/CD-Pipelines

André Kirsch, André Matutat, Malte Reinsch, Birgit Christina George und Carsten Gips[1]

Abstract: Im Rahmen des Moduls Programmieren 2 des Studiengangs Informatik BA der Hochschule Bielefeld reichen Studierende seit mehreren Jahren ihre Programmierlösungen als Git-Pull-Requests ein und verlinken diese nur noch in ihrer Abgabe im Learning Management System. Bisher wurden die Lösungen der Studierenden anschließend in Präsenz mit den Lehrenden diskutiert und von diesen bewertet. Da diese Art der Bewertung viel Zeit in Anspruch nimmt, arbeiten wir an der Umstellung auf ein automatisches Bewertungssystem. Aus diesem Grund präsentieren wir in diesem Paper unser Konzept zur automatischen Bewertung von Programmieraufgaben mithilfe von Continuous Integration/Continuous Deployment-Pipelines. Im Gegensatz zu anderen automatischen Bewertungssystemen verwenden wir keine eigene Serverstruktur, sondern nutzen frei verfügbare Infrastruktur. Wir berücksichtigen dabei die einfache Übertragung auf andere Continuous Integration/Continuous Deployment-Pipelines sowie die Möglichkeit zur lokalen Ausführung.

Keywords: Autograding; Automatisiertes Feedback; CI/CD-Pipeline

1 Einleitung

Studierende erhalten in Programmiermodulen häufig umfangreiche praktische Aufgaben, um gelernte Inhalte anzuwenden. Um die Lösungen zu den Aufgaben zu bewerten, müssen Lehrende diese kontrollieren, was ein stark repetitiver Vorgang ist. Die manuelle Bewertung des Programmcodes erfordert häufig viel Zeit, was den Lehrenden die Möglichkeit nimmt, qualitative Probleme mit den Studierenden zu diskutieren oder individuelle Beratungen durchzuführen.

Im Rahmen des Moduls Programmieren 2 im Studiengang Informatik BA an der Hochschule Bielefeld möchten wir ein automatisches Bewertungssystem für Java entwickeln und einsetzen. Bei der Bearbeitung der Programmieraufgaben wird Git eingesetzt. Studierende bearbeiten die Aufgaben lokal und laden ihre Lösungen in ihr Team-Repository auf einem Git-Server (GitHub/GitLab/...). Dort erstellen sie zur Abgabe einen Pull Request (PR), welchen sie anschließend im Learning Management System (LMS) ILIAS verlinken.

Im Rahmen dieses Papers möchten wir ein Konzept vorstellen, durch das wir den bisherigen Abgabeprozess mit einer automatischen Bewertung verknüpfen. Studierende sollen weiterhin einen PR zur Abgabe erstellen. Commits zu diesem PR sollen automatisch eine Continuous Integration/Continuous Deployment (CI/CD)-Pipeline (hier konkret eine GitHub Action) ausführen, die den Bewertungsprozess durchführt. Studierende können in der Ausgabe der GitHub Action ihre erreichte Punktzahl einsehen. Ein Ziel des vorgestellten Konzeptes

[1] Hochschule Bielefeld, Campus Minden, Artilleriestraße 9, 32427 Minden, Deutschland, andre.kirsch@hsbi.de

doi:10.18420/abp2023-11

für ein automatisches Bewertungssystem soll die Zeitersparnis für Lehrende bei der Beurteilung von Programmierlösungen sein, wobei die Rückmeldungen bei der Begründung von Punktabzügen ausreichend ausführlich bleiben sollten. So bleibt für die Lehrenden mehr Zeit für die fachliche Diskussion mit Studierenden. Im Gegensatz zu anderen automatischen Bewertungssystemen nutzen wir keine eigene Serverstruktur und lehren gleichzeitig den Umgang mit Git und Free Open Source Software (FOSS). Daher ist unser automatisches Bewertungssystem eher für fortgeschrittenere Programmierkurse ab dem zweiten Semester geeignet. Die Implementierung des Konzepts soll in den nächsten Monaten stattfinden, sodass das System im Sommersemester 2024 erstmals eingesetzt und evaluiert werden kann.

Zusammenfassend ist der wesentliche Inhalt dieses Papers ein neues Konzept für die automatische Bewertung von Programmierlösungen unter Verwendung eines Versionskontrollsystems und einer CI/CD-Pipeline. Da die Implementierung des Konzeptes noch nicht abgeschlossen ist, wird nicht auf die konkrete Implementierung eingegangen und es können auch noch keine Erfahrungen zum Einsatz des automatischen Bewertungssystems präsentiert werden.

Im weiteren Verlauf des Papers werden in Kapitel 2 verwandte Arbeiten vorgestellt, wobei wir unser Konzept in das Themengebiet einordnen. Anschließend folgt in Kapitel 3 ein Überblick über unser aktuelles Abgabeverfahren. Dabei wird insbesondere auch darauf eingegangen, warum wir bei unserem automatischen Bewertungsverfahren einen Git-basierten Ansatz gewählt haben. In Kapitel 4 folgt eine detaillierte Erläuterung des erarbeiteten Konzepts. Das letzte Kapitel 5 schließt das Paper ab und gibt einen Ausblick auf die Umsetzung und den Einsatz des Deploy-to-Grading (D2G).

2 Verwandte Arbeiten

Automatische Bewertungssysteme werden schon seit langem international sowie in der deutschen Universitäts- und Hochschullandschaft genutzt, um mit wenig Zeitaufwand Programmierlösungen von Studierenden zu prüfen und zu bewerten. Beispiele sind ASB [HOS17] der Hochschule Trier, JACK [St17] von der Universität Duisburg-Essen und CodeOcean [St16] des Hasso-Plattner-Instituts, die vor einigen Jahren entwickelt wurden und stetig verbessert und erweitert werden. CodeOcean ist ein Autograder, mit dem Studierende durchgehend online arbeiten. Er stellt die Aufgaben zur Verfügung, die in einem Online-Editor gelöst werden. Das automatische Bewerten wird von derselben Weboberfläche aus ausgeführt. ASB und JACK kennzeichnen sich auch durch eine eigene Client-Server-Struktur. Die Lösungen werden von den Studierenden lokal erarbeitet und anschließend in die Webanwendung für die Bewertung hochgeladen. D2G soll im Gegensatz zu den genannten Autogradern keine eigene Client-Server-Struktur benötigen.

Neben den automatischen Bewertungssystemen hat der Git-Arbeitsablauf insbesondere in fortgeschrittenen Lehrmodulen Einzug gehalten, weswegen die Nutzung von Git in einem automatischen Bewertungssystem sinnvoll erscheint. GitGrade [Zh20] ist ein solches

Git-basiertes automatisches Bewertungssystem, welches an der University of Washington entwickelt wurde und auf einer universitätseigenen GitLab-Instanz aufsetzt. Es beinhaltet typische Bewertungsverfahren und unterstützt qualitatives Code Review-Feedback. Studierende können über eine eigene Weboberfläche Aufgabenstellungen akzeptieren, die sie in einem GitLab Repository bearbeiten und anschließend über dieselbe Weboberfläche einreichen können. Im Gegensatz zu GitGrade und anderen Autogradern soll D2G keine eigene Serverinstanz benötigen, sondern ausschließlich lokal oder als CI/CD-Pipeline ausgeführt werden. Dabei liegt ein Augenmerk auf der Portierbarkeit.

GitHub bietet mit GitHub Classroom [Gi23] selbst ein kostenfreies Tool zum Bewerten von Programmierlösungen an. Jede Aufgabe wird dabei in einem eigenen Repository verwaltet. Über ein Webinterface kann dieses Repository in eine Aufgabenstellung umgewandelt werden. Über einen Link können Studierende die Bearbeitung der Aufgabe starten. GitHub Classroom vereinfacht insbesondere die Verteilung der Aufgaben. Außerdem zeigt es automatisch eine Übersicht über die Lösungen an. Ein Nachteil von GitHub Classroom ist die fehlende Flexibilität bei der Bepunktung der Lösungen, da Teilpunkte bei Tests nicht vergeben werden können. Weitere Negativpunkte sind das Fehlen einer Repository übergreifenden Plagiatsprüfung sowie der Möglichkeit zur lokalen Ausführung. Des Weiteren versteckt es den Git-Workflow in großen Teilen. In unserem Konzept möchten wir die Flexibilität bei der Bepunktung beibehalten sowie eine Plagiatsprüfung durchführen können.

3 Bisheriges Vorgehen

Im Modul Programmieren 2 setzen die Studierenden aktiv Git und einen beliebigen Git-basierten Workflow für die Zusammenarbeit in Dreier-Teams ein. Dazu erstellen sie sich zuerst einen Fork des Vorgabe-Repositorys. Häufig ist dies für die Studierenden der erste Berührungspunkt mit Git und einer der verschiedenen Online-Plattformen wie GitHub oder GitLab. Die Studierenden arbeiten gemeinsam an einem Projekt und teilen ihren Arbeitsfortschritt innerhalb des Teams. Für die Abgabe erstellen sie einen öffentlich zugreifbaren PR innerhalb ihres Repositorys und geben die URL zu diesem PR im LMS als Link ab. Das Lösungskonzept und die Umsetzung (Code) werden im Praktikum den Lehrenden vorgestellt und von diesen bewertet.

Die Verwendung von Git hat mehrere Vorteile im Vergleich zur Abgabe von Lösungen im LMS. So erlernen Studierende tiefergehend den typischen Arbeitsablauf in Unternehmen und FOSS-Projekten. In PRs können die konkreten Änderungen eingesehen und direkt Feedback am Quellcode gegeben werden. Auch wird verhindert, dass unvollständige Projektordner abgegeben werden. Nach Kertész [Ke15] müssen sich Studierende initial erst an die Arbeitsweise gewöhnen, können dadurch aber nicht nur ihre Programmierfähigkeiten, sondern auch ihre Gruppenarbeitskompetenz verbessern. Möglich ist auch, dass Studierende aktiv an der Verbesserung des Lehrmaterials mithelfen, wenn dieses in einem Repository öffentlich verfügbar ist. Ein Nachteil dieses Verfahrens ist, dass für die Benotung ein Bezug zwischen einem GitHub-Account und einer reellen Person hergestellt werden muss. Aus

Gründen des Datenschutzes sollte dies nicht über GitHub geschehen, weswegen trotzdem ein LMS notwendig ist, über das in unserem Fall der PR verlinkt wird.

4 Deploy-to-Grading

Beim D2G soll der bisherige Arbeitsablauf von Studierenden zu großen Teilen beibehalten werden. Studierende arbeiten weiterhin in einem Fork des Repositorys, in dem die Aufgabenstellungen liegen. Sie bearbeiten diese Aufgaben in einer frei wählbaren IDE und laden ihre Änderungen über Git hoch. Zur Abgabe erstellen sie einen PR. Dabei und bei jedem weiteren Commit zu diesem PR wird D2G automatisiert gestartet. Dadurch, dass das D2G auch bei jedem weiteren Commit ausgeführt wird, haben Studierende bis zur Deadline die Möglichkeit, ihre Lösung zu überarbeiten und erneut prüfen zu lassen. Einen Überblick darüber verschafft Abbildung 1. Damit zielen wir auch darauf ab, den typischen Softwareentwicklungsprozess mit CI/CD im Unternehmens- und FOSS-Kontext abzubilden. Gleichzeitig soll unser System offen und modular sein, sodass auch andere Programmiersprachen und Bewertungsmetriken verwendet werden können.

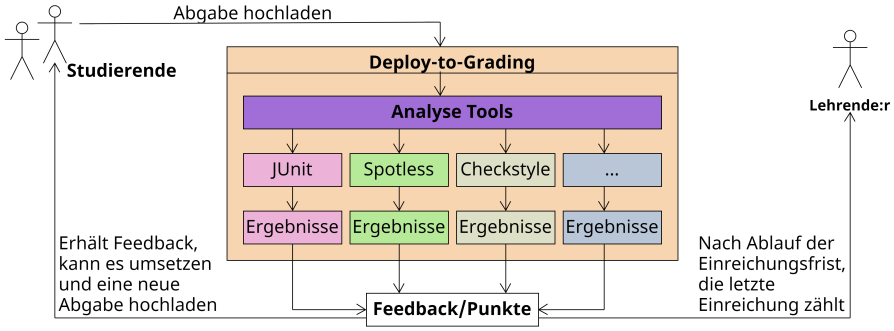


Abb. 1: Ablauf des Abgabeverfahrens von D2G. Studierende laden ihre Abgabe hoch, indem sie einen PR erstellen. Dadurch wird automatisch das D2G gestartet. Dieses führt für eine Aufgabe alle erforderlichen Tools aus und fasst dessen Ergebnisse zusammen. Daraus wird für die Studierenden ein Feedback und eine Punkteübersicht generiert. Studierende können bis zur Einreichungsfrist den Vorgang unbegrenzt wiederholen. Nach Ablauf der Einreichungsfrist kann der/die Lehrende die Ergebnisse einsammeln und für die Benotung nutzen.

Das D2G steht als GitHub Action in einem öffentlichen Repository zur Verfügung und wird über einen GitHub Workflow in das Studierenden-Repository eingebunden. Bei der Wahl der Onlineplattform und der eingesetzten Analysetools möchten wir flexibel sein und außerdem die lokale Ausführung ermöglichen. Durch dieses Konzept können Studierende und Lehrende auf eigene Ressourcen ausweichen (z. B. bei Datenschutzbedenken).

D2G checkt das zu prüfende Repository lokal aus und analysiert die relevanten Commits im Abgabeintervall. Anschließend wird das D2G für jede konfigurierte Aufgabenstellung

ausgeführt. Dabei werden zunächst alle Vorgabedateien mit den Dateien aus dem Vorgabe-Repository überschrieben, um ein Ändern dieser Dateien zu verhindern. Ausnahmen können hierbei in der Konfiguration angegeben werden. Anschließend folgt die Ausführung der Bewertungsmetriken. Bewertungsmetriken können dabei flexibel zu jeder Aufgabenstellung hinzugefügt und umfangreich konfiguriert werden. Die Ergebnisse werden in einem *results*-Ordner gesammelt, um sie zuletzt aufbereitet in der Konsole auszugeben und als Artefakt zur Verfügung zu stellen.

In den folgenden Unterkapiteln wird detaillierter auf einzelne Teile des D2G eingegangen. Dabei wird zuerst in Unterkapitel 4.1 ein Überblick über die Repository- und Dateistrukturen gegeben. Anschließend folgt in Unterkapitel 4.2 ein Überblick über die angewandten Bewertungsmetriken. Das Unterkapitel 4.3 beschreibt, wie die Ergebnisse des D2G den Studierenden und den Lehrenden übermittelt werden. Grenzen des Konzepts werden in Kapitel 4.4 aufgezeigt.

4.1 Aufbau der Repository- und Dateistrukturen

Da das D2G primär auf einem Git-Server genutzt werden soll, ist es naheliegend, auch die Aufgaben darüber zu verwalten. Wie in Abbildung 2 dargestellt, werden in einem nicht öffentlich sichtbaren Repository (*HomeworkSolution*) Aufgabenstellungen, Vorgabecode und Beispiellösungen gepflegt. Über *git subtree* können die Aufgabenstellungen und der Vorgabecode in ein für die Studierenden zugreifbares Repository *Template* exportiert werden. Es ist optional auch möglich, über *git filter-repo* einzelne Aufgaben in separate *Tasks*-Repositorys zu exportieren. Die Studierende erstellen vom *Template*- oder den *Tasks*-Repository(s) einen Fork, in dem sie die Aufgaben lösen und über den sie ihre Lösungen einreichen. Das *Template*- und alle darunter liegenden Repositorys sind mit unserem GitHub Workflow konfiguriert, welcher das D2G ausführt. Der Workflow verweist auf das *D2G*-Repository, welches das D2G als GitHub Action implementiert.

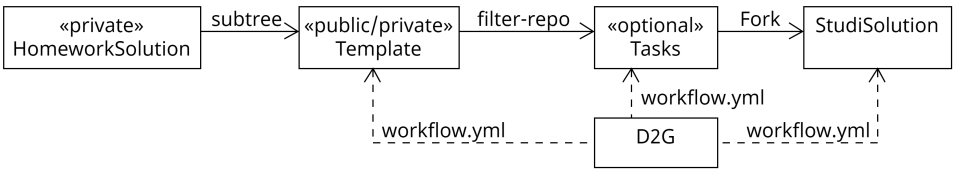


Abb. 2: Neben dem *D2G*-Repository für die D2G-GitHub-Action (CI/CD-Pipeline) existieren weitere Repositorys, die das D2G konfigurieren. Diese enthalten eine oder mehrere Aufgabenstellungen mit Vorgabecode. Im *HomeworkSolution*-Repository sind zusätzlich Beispiellösungen vorhanden. Das *StudiSolution*-Repository steht stellvertretend für alle Repositorys von Studierenden.

Für den Einsatz des D2G werden Konfigurationsdateien benötigt, die im selben Repository abgelegt werden wie die Aufgaben. Dazu gehört eine Konfigurationsdatei für das Aufgabenblatt sowie für jede Aufgabe eine eigene Konfigurationsdatei, in der die Bewertungsmetriken

konfiguriert werden. Die hier aufgezählten Metriken werden vom D2G ausgeführt. Alternativ prüfen wir die Unterstützung etablierter Konfigurationsformate, wie beispielsweise ProFormA 2.0 [Re19] und PEML [ME23].

4.2 Bewertungsmetriken

Bei der Auswahl an Bewertungsmetriken orientieren wir uns an typischen Bewertungsmaßstäben für Java-Quellcode und unterstützen unter anderem die statische sowie dynamische Codeanalyse. Dazu gehören beispielsweise die erfolgreiche Kompilierung, Unittests, CLI-Tests, Coding-Style, Dokumentation und Plagiatsprüfung. Des Weiteren planen wir die Umsetzung und Evaluation weiterer Blackbox-Testing-Möglichkeiten. Das D2G wird so umgesetzt, dass neue Bewertungsmetriken, insbesondere auch für andere Programmiersprachen, relativ einfach hinzugefügt werden können. Hervorzuheben ist, dass dieses Konzept mit entsprechenden Analysetools auch für Aufgabenstellungen jenseits der Informatik, z. B. für die Analyse von geschriebenen Texten, eingesetzt werden kann.

4.3 Rückmeldung an Studierende und Lehrende

Die Möglichkeiten zur Ausgabe der Ergebnisse an Studierende ist aufgrund der nicht genutzten eigenen Serverstruktur leicht eingeschränkt. D2G generiert eine Ausgabe der erreichten Punkte und einen Überblick über die Punktabzüge als Konsolenausgabe des GitHub Workflows. Da die Übersicht hier begrenzt ist, arbeiten wir außerdem an der Generierung eines Reports, der die Ergebnisse visuell aufbereitet. Zusätzlich werden die detaillierten Log-Files der Analysetools als Artefakt im GitHub Workflow den Studierenden zugänglich gemacht.

Lehrende können für das Erstellen einer Ergebnisübersicht D2G lokal ausführen und so die Ergebnisse der Studierenden datenschutzgerecht einsammeln und auswerten. Dazu wird der Anwendung eine aus dem LMS geladene Liste mit PR-URLs übergeben. Über die GitHub-API werden die Ergebnisse der PRs der Studierenden eingesammelt. Alternativ ist es auch möglich, den Bewertungsprozess erneut lokal auszuführen. Dies ist notwendig, wenn die Verfügbarkeit eines Artefakts abgelaufen ist. Zusätzlich kann eine Plagiatsprüfung durchgeführt werden. Abschließend wird eine *csv*-Datei mit den Punkten generiert, welche zurück in das LMS überführt werden muss.

4.4 Grenzen

Mit den Anforderungen der Verwendung öffentlich verfügbarer Infrastruktur und dem Git-basierten Workflow besitzt D2G Grenzen, die bei anderen automatischen Bewertungssystemen nicht zu finden sind. So ist man bei der Nutzung von D2G stark abhängig von der

öffentlich verfügbaren Infrastruktur, auf die man keinen Einfluss hat. Das System ist zwar so konzipiert, dass ein Wechsel zu anderen Anbietern leicht möglich sein soll, aber das kann mit zusätzlichem Arbeitsaufwand einhergehen. Außerdem besitzen GitHub, GitLab, usw. eine Begrenzung bei der Nutzung der Infrastruktur. Umgangen werden kann dieses Problem durch die Nutzung einer eigenen Server-Infrastruktur (z. B. eigene GitLab-Instanz).

Ein weiteres Problem sind die in der Regel öffentlich zugänglichen Studierenden-Repositorys, wodurch ein Kopieren der Lösungen von anderen Studierenden nicht ausgeschlossen werden kann. Zwar existieren Lösungen, um private Repositorys zu erstellen, dabei tauchen aber andere Probleme auf (z. B. erschwerter Zugriff durch die Lehrperson). Die eingesetzte Plagiatsprüfung sowie der öffentliche einsehbare Commit-Verlauf stellen zwar eine größere Hemmschwelle zum Kopieren von Lösungen dar, ob diese tatsächlich ausreichen, muss aber eine Evaluation zeigen. Möglicherweise ist auch eine Individualisierung der Aufgabenstellungen notwendig. Weiterhin ist aufgrund des Git-basierten Workflows ein Einsatz des D2G nur möglich, wenn im selben Modul oder in einem vorherigen Semester Git gelehrt wird, da das Abgabeverfahren ansonsten eine zu große Hürde für Studierende darstellen könnte.

Bei der Gestaltung von Übungsaufgaben ist im Vergleich zu Aufgabenstellungen ohne automatische Bewertung für die automatische Bewertung zusätzlicher Aufwand notwendig. Dazu gehören zum Beispiel die Erstellung von Unittests und die Bereitstellung von konkreten Vorgaben. Dabei ist darauf zu achten, dass die Vorgaben und Tests derart gestaltet werden, dass einerseits die automatische Testbarkeit gewährleistet ist, andererseits die Anzahl möglicher Lösungswege nicht zu sehr eingeschränkt wird.

5 Zusammenfassung und Ausblick

In diesem Paper haben wir mit D2G ein modulares und flexibles Konzept zur automatischen Bewertung von Programmierlösungen unter Gewährleistung des Datenschutzes vorgestellt. Es basiert auf dem Git-Workflow und nutzt CI/CD-Pipelines (GitHub Actions) zur Ausführung. Im Gegensatz zu anderen Lösungen verwenden wir keine eigene Serverstruktur, sondern nutzen frei zur Verfügung stehende Ressourcen. Des Weiteren bleiben wir mit der Umsetzung flexibel und ermöglichen somit die einfache Portierung (z. B. als Docker Container) auf unterschiedliche Git-Server sowie auch die lokale Ausführung. Gleichzeitig ermöglichen wir mit dem modularen Ansatz für Bewertungsmetriken die Erweiterung auf andere Programmiersprachen und Domänen.

Mit der Implementierung des vorgestellten Konzepts wurde bereits im Sommersemester 2023 begonnen. Wir planen, im darauf folgenden Wintersemester die Implementierung fertigzustellen und den praktischen Einsatz vorzubereiten (Überarbeitung der Aufgaben, Implementierung von Tests). Die neu entwickelten Aufgabenstellungen sollen sich spezifisch mit bestimmten Vorlesungsthemen auseinandersetzen. Zusätzlich sollen dann auch die Vorlesungsthemen in Sinne des Self-Paced-Learnings in einer relativ freien Reihenfolge bearbeitet werden können - das automatische Bewertungssystem soll diesen Spielraum

ermöglichen. Es soll im Sommersemester 2024 erstmalig produktiv eingesetzt und auch in der Lehrveranstaltung evaluiert werden. In der Evaluation möchten wir weitere Vor- und Nachteile des vorgestellten automatischen Bewertungssystems für die Lehrenden und insbesondere Studierenden ermitteln und prüfen, welchen Einfluss das System auf den Lernerfolg hat. Außerdem möchten wir umfangreiche Praxiserfahrung sammeln und das System weiter verfeinern. Der aktuelle Stand des Projektes befindet sich auf GitHub unter `https://github.com/Programmiermethoden/Deploy-to-Grading`.

Literaturverzeichnis

[Gi23] GitHub Inc.: GitHub Classroom, `https://classroom.github.com`, [Letzter Zugriff: 02.06.2023], 2023.

[HOS17] Herres, B.; Oechsle, R.; Schuster, D.: Der Grader ASB. In (Bott, O. J.; Fricke, P.; Priss, U.; Striewe, M., Hrsg.): Automatisierte Bewertung in der Programmierausbildung. Kap. 16, 2017, ISBN: 978-3-8309-3606-0.

[Ke15] Kertész, C.-Z.: Using GitHub in the classroom - a collaborative learning experience. In: IEEE 21st International Symposium for Design and Technology in Electronic Packaging (SIITME). S. 381–386, 2015.

[ME23] Mishra, D. S.; Edwards, S. H.: The Programming Exercise Markup Language: Towards Reducing the Effort Needed to Use Automated Grading Tools. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1. S. 395–401, 2023.

[Re19] Reiser, P.; Borm, K.; Feldschnieders, D.; Garmann, R.; Ludwig, E.; Müller, O.; Priss, U.: ProFormA 2.0 – ein Austauschformat für automatisiert bewertete Programmieraufgaben und für deren Einreichungen und Feedback. In: 4. Workshop „Automatische Bewertung von Programmieraufgaben“. S. 43–50, 2019.

[St16] Staubitz, T.; Klement, H.; Teusner, R.; Renz, J.; Meinel, C.: CodeOcean - A versatile platform for practical programming excercises in online environments. In: IEEE Global Engineering Education Conference. S. 314–323, 2016.

[St17] Striewe, M.: Der Grader JACK. In (Bott, O. J.; Fricke, P.; Priss, U.; Striewe, M., Hrsg.): Automatisierte Bewertung in der Programmierausbildung. Kap. 9, 2017, ISBN: 978-3-8309-3606-0.

[Zh20] Zhang, J. K.; Lin, C. H.; Hovik, M.; Bricker, L. J.: GitGrade: A Scalable Platform Improving Grading Experiences. In: Proceedings of the 51st ACM Technical Symposium on Computer Science Education. 2020.